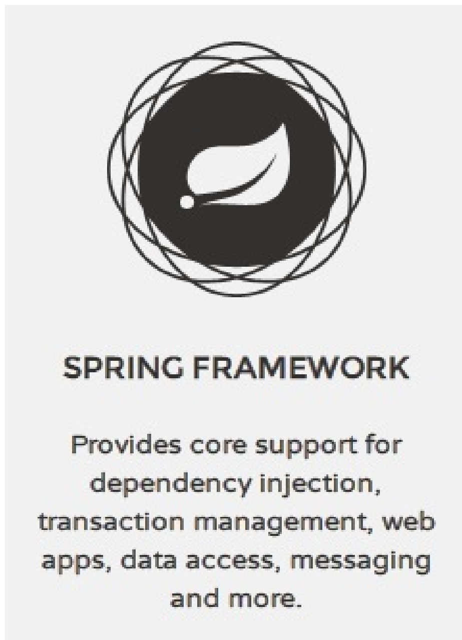
Table of Contents

spring-framework-4-reference



Chinese translation of [Spring Framework 4.x Reference Documentation] (https://docs.spring.io/spring/docs/4.3.13.RELEASE/spring-framework-reference/html/). The current version of Spring Framework 4.x is 4.3.9.RELEASE. There is also a GitBook version of the book: http://waylau.gitbooks.io/spring-framework-4-reference. Let's READ!

《Spring Framework 4.x参考文档》中文翻译（包含了官方文档以及其他文章）。至今为止，Spring Framework 的最新版本为 4.3.13.RELEASE。

利用业余时间对此进行翻译，并在原文的基础上，插入配图，图文并茂方便用户理解。如有勘误欢迎指正，点此提问。如有兴趣，也可以参与到本翻译工作中来 :)

Get start 如何开始阅读

选择下面入口之一：

- https://github.com/waylau/spring-framework-4-reference 的 SUMMARY.md（源码）
- http://waylau.gitbooks.io/spring-framework-4-reference 的 Read 按钮（同步更新，国内访问速度一般）

Issue 意见、建议

如有勘误、意见或建议欢迎拍砖 https://github.com/waylau/spring-framework-4-reference/issues

Contact 联系作者:

- Blog: waylau.com
- Gmail: waylau521(at)gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

Part I. Overview of Spring Framework 总览

Spring Framework 是一个轻量级的解决方案，可以一站式构建企业级应用。然而，Spring 是模块化的，允许你使用的你需要的部分，而不必把其余带进来。你可以在任何框架之上去使用IOC容器，但你也可以只使用 Hibernate 集成代码 或 JDBC 抽象层。Spring Framework 支持声明式事务管理，通过 RMI 或 Web 服务远程访问你的逻辑，并支持多种选择持久化你的数据。它提供了一个全功能的 MVC 框架，使您能够将 AOP 透明地集成到您的软件。

Spring 的设计是非侵入性的，也就是说你的领域逻辑代码一般对框架本身无依赖性。在你的集成层（如数据访问层），在数据访问技术和 Spring 的库一些依赖将存在。然而，它应该很容易从你的剩余代码中分离这些依赖。

本文档是 Spring Framework 功能的参考指南。如果你有关于这个文档的任何要求，意见或问题，请发送到用户邮件列表。对 Framework 本身的问题应该到 StackOverflow 请问（见 https://spring.io/questions）

译者注：对本文档的翻译有任何问题，请在https://github.com/waylau/spring-framework-4-reference/issues上面提问

开始

本参考指南提供了关于 Spring Framework 的详细信息。提供它的所有功能全面的文档，以及 Spring 所涵盖的一些关于底层方面的背景资料（如“Dependency Injection（依赖注入）”）。

如果你是刚刚开始接触 Spring，你可能要开始使用 Spring Framework 创建一个基于 Spring Boot 的应用。Spring Boot 提供了一种快速、自动配置 Spring 各种组件的方法来创建一个用于生产环境的 Spring 的应用程序。它是基于 Spring Framework，约定大于配置，目的是快速搭建一个可以运行的应用。

您可以使用 start.spring.io 创建一个基本项目或遵循一个类似于Getting Started Building a RESTful Web Service的“入门”指南。这些指南都非常专注于具体的任务，其中大部分是基于 Spring Boot，非常容易理解。这些还涵盖了你可能想解决一个特定问题时要考虑到的其他 Spring 项目。

介绍 Spring Framework

Spring Framework 是一个提供完善的基础设施用来支持来开发 Java 应用程序的 Java 平台。Spring 负责基础设施功能，而您可以专注于您的应用。

Spring 可以使你从“简单的Java对象”（POJO）构建应用程序，并且将企业服务非侵入性的应用到 POJO。此功能适用于 Java SE 编程模型和完全或者部分的 Java EE 。

举例，作为一个应用程序的开发者，你可以从 Spring 平台获得以下好处：

- 使 Java 方法可以执行数据库事务而不用去处理事务 API。
- 使本地 Java 方法可以执行远程过程而不用去处理远程 API。
- 使本地 Java 方法可以拥有管理操作而不用去处理 JMX API。
- 使本地 Java 方法可以执行消息处理而不用去处理 JMS API。

依赖注入和控制反转

Java 应用程序--运行在各个松散的领域,从受限的嵌入式应用程序,到 n 层架构的服务端企业级应用程序--通常由来自应用适当的对象进行组合合作。因此,对象在应用程序中是彼此依赖。

尽管 Java 平台提供了丰富的应用程序开发功能,但它缺乏来组织基本构建块成为一个完整的方法。这个任务留给了架构师和开发人员 。虽然您可以使用设计模式,例如 Factory, Abstract Factory, Builder, Decorator, 和 Service Locator 来组合各种类和对象实例构成应用程序,这些模式是:给出一个最佳实践的名字,描述什么模式,哪里需要应用它,它要解决什么问题,等等。模式是形式化的最佳实践,但你必须在应用程序中自己来实现。

Spring Framework 的 *Inversion of Control* (IoC) 组件旨在通过提供正规化的方法来组合不同的组件成为一个完整的可用的应用。 Spring Framework 将规范化的设计模式作为一等的对象,您可以集成到自己的应用程序。许多组织和机构使用 Spring Framework 以这种方式来开发健壮的、可维护的应用程序。

控制反转和依赖注入的背景

“问题是,*[他们]*哪些方面的控制被反转?”这个问题由 *Martin Fowler*在他的 *Inversion of Control (IoC)* 网站在 *2004* 年提出。 *Fowler* 建议重新命名这个说法,使得他更加好理解,并且提出了 *Dependency Injection*(依赖注入) 这个新的说法。

译者注:Dependency Injection 和 Inversion of Control 其实就是一个东西的两种不同的说法而已。本质上是一回事。Dependency Injection 是一个程序设计模式和架构模型, 一些时候也称作 Inversion of Control,尽管在技术上来讲,Dependency Injection 是一个 Inversion of Control 的特殊实现,Dependency Injection 是指一个对象应用另外一个对象来提供一个特殊的能力,例如:把一个数据库连接以参数的形式传到一个对象的结构方法里面而不是在那个对象内部自行创建一个连接。Inversion of Control 和 Dependency Injection 的基本思想就是把类的依赖从类内部转化到外部以减少依赖。 应用Inversion of Control,对象在被创建的时候,由一个调控系统内所有对象的外界实体,将其所依赖的对象的引用,传递给它。也可以说,依赖被注入到对象中。所以,Inversion of Control 是,关于一个对象如何获取他所依赖的对象的引用,这个责任的反转。

模块

Spring Framework 的功能被组织成了 20 来个模块。这些模块分成 Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, Messaging, 和 Test，如下图：

Figure 2.1. Overview of the Spring Framework



下面章节会列出可用的模块，名称与功能及他们的主题相关。组件名称与组件的 ID 相关，用于2.3.1节中的依赖管理工具。

核心容器

Core Container 由 spring-core, spring-beans, spring-context, spring-context-support, 和 spring-expression (Spring Expression Language) 模块组成。

spring-core 和 spring-beans 提供框架的基础部分，包括 IoC 和 Dependency Injection 功能。BeanFactory 是一个复杂的工厂模式的实现。不需要可以编程的单例，并允许您将配置和特定的依赖从你的实际程序逻辑中解耦。

Context (spring-context) 模块建立且提供于在Core 和 Beans 模块的基础上，它是一种在框架类型下实现对象存储操作的手段，有一点像 JNDI 注册。Context 继承了 Beans 模块的特性，并且增加了对国际化的支持（例如用在资源包中）、事件广播、资源加载和创建上下文（例如 一个 Servlet 容器）。Context 模块也支持例如 EJB，JMX 和基础远程这样的 JavaEE 特性。ApplicationContext 是 Context 模块的焦点。spring-context-support 提供对常见第三方库的支持集成进 Spring 应用上下文，如缓存 (EhCache, Guava, JCache), 通信 (JavaMail), 调度 (CommonJ, Quartz) 和 模板引擎 (FreeMarker, JasperReports, Velocity)。

spring-expression 模块提供了一个强大的Expression Language（表达式语言）用来在运行时查询和操作对象图。这是作为JSP2.1 规范所指定的统一表达式语言（unified EL）的一种延续。这种语言支持对属性值、属性参数、方法调用、数组内容存储、收集器和索引、逻辑和算数操作及命名变量，并且通过名称从 Spring 的控制反转容器中取回对象。表达式语言模块也支持 List 的映射和选择，正如像常见的列表汇总一样。

AOP 及 Instrumentation

spring-aop 模块提供 AOP Alliance-compliant（联盟兼容）的面向切面编程实现，允许你自定义，比如，方法拦截器和切入点完全分离代码。使用源码级别元数据的功能，你也可以在你的代码中加入 behavioral information (行为信息)，在某种程度上类似于 .NET 属性。

单独的 spring-aspects 模块提供了集成使用 AspectJ。

spring-instrument 模块提供了类 instrumentation 的支持和在某些应用程序服务器使用类加载器实现。spring-instrument-tomcat 用于 Tomcat Instrumentation 代理。

消息

Spring Framework 4 包含了 spring-messaging 模块，从 Spring 集成项目中抽象出来，比如 Message, MessageChannel, MessageHandler 及其他用来提供基于消息的基础服务。该模块还包括一组消息映射方法的注解，类似于基于编程模型 Spring MVC 的注解。

数据访问/集成

Data Access/Integration 层由 JDBC, ORM, OXM, JMS, 和 Transaction 模块组成。

spring-jdbc 模块提供了不需要编写冗长的JDBC代码和解析数据库厂商特有的错误代码的 JDBC-抽象层。

spring-tx 模块的支持可编程和声明式事务管理，用于实现了特殊的接口的和你所有的POJO类（Plain Old Java Objects）。

spring-orm 模块提供了流行的 object-relational mapping（对象-关系映射）API集成层，其包含 JPA，JDO，Hibernate。使用ORM包，你可以使用所有的 O/R 映射框架结合所有Spring 提供的特性，比如前面提到的简单声明式事务管理功能。

spring-oxm 模块提供抽象层用于支持 Object/XML mapping （对象/XML映射）的实现,如 JAXB、Castor、XMLBeans、JiBX 和 XStream 等。

spring-jms 模块（Java Messaging Service）包含生产和消费信息的功能。 从 Spring Framework 4.1 开始提供集成 spring-messaging 模块。

2.2.5 Web

Web 层由 spring-web, spring-webmvc, spring-websocket, 和 spring-webmvc-portlet 组成 。

spring-web 模块提供了基本的面向 web 开发的集成功能，例如多方文件上传、使用 Servlet listeners 和 Web 开发应用程序上下文初始化 IoC 容器。它也包含 HTTP 客户端以及 Spring 远程访问的支持的 web 相关的部分。

spring-webmvc 模块（也被称为 Web Servlet 模块）包含 Spring 的model-view-controller（模型-视图-控制器（MVC）和 REST Web Services 实现的Web应用程序。Spring 的 MVC 框架提供了domain model（领域模型）代码和 web form (网页) 之间的完全分离，并且集成了 Spring Framework 所有的其他功能。

spring-webmvc-portlet 模块（也被称为 Web-Portlet 模块）提供了MVC 模式的实现是用一个 Portlet 的环境和 spring-webmvc 模块功能的镜像。

2.2.6 Test

spring-test 模块支持通过组合 JUnit 或 TestNG 来进行单元测试和集成测试 。它提供了连续的加载 ApplicationContext 并且缓存这些上下文。它还提供了 mock object（模仿对象），您可以使用隔离测试你的代码。

使用场景

前面的构建模块描述在很多的情况下使 Spring 是一个合理的选择，从资源受限的嵌入式程序到成熟的企业应用程序都可以使用 Spring 事务管理功能和 web 框架集成。

Figure 2.2. Typical full-fledged Spring web application



Spring 声明式事务管理特性使 Web 应用程序拥有完全的事务，就像你使用 EJB 容器管理的事务。所有的自定义业务逻辑可以用简单的 POJO 实现，用 Spring 的 IoC 容器进行管理。额外的服务包括发送电子邮件和验证，是独立的网络层的支持，它可以让你选择在何处执行验证规则。Spring 的 ORM 支持集成 JPA，Hibernate，JDO；例如，当使用 Hibernate，您可以继续使用现有的映射文件和 标准的 Hibernate 的 SessionFactory 配置。表单控制器将 Web 层和领域模型无缝集成，消除 ActionForms 或其他类用于变换 HTTP 参数成为您的域模型值的需要。

Figure 2.3. Spring middle-tier using a third-party web framework



有时，不允许你完全切换到一个不同的框架。Spring Framework 不强制使用它,它不是一个全有或全无的解决方案。现有的前端 Struts, Tapestry, JSF 或其他 UI 框架，可以集成一个基于 Spring 中间件，它允许你使用 Spring 事务的功能。你只需要将业务逻辑连接使用 ApplicationContext 和使用WebApplicationContext 集成到你的 web 层。

Figure 2.4. Remoting usage scenario



当你需要通过 web 服务访问现有代码时，可以使用 Spring 的 Hessian-, Burlap-, Rmi- 或 JaxRpcProxyFactory 类。启用远程访问现有的应用程序并不难。

Figure 2.5. EJBs - Wrapping existing POJOs



Spring Framework 也提供了 Enterprise JavaBeans 访问和抽象层 使你能重用现有的 POJOs, 并且在需要声明安全的时候打包他们成为无状态的 bean 用于可伸缩，安全的 web 应用里。

依赖关系管理和命名约定

依赖管理和依赖注入是不同的概念。为了让 Spring 的这些不错的功能运用到运用程序中（比如依赖注入），你需要收集所有的需要的库（JAR文件），并且在编译、运行的时候将它们放到你的类路径中。这些依赖不是虚拟组件的注入，而是物理的资源在文件系统中（通常）。

依赖管理过程包括定位这些资源，存储它们并添加它们到类路径。依赖可以是直接（如我的应用程序运行时依赖于 Spring ），或者是间接（如我的应用程序依赖 commons-dbcp ，而 commons-dbcp 又依赖于 commons-pool）。间接的依赖也被称为 “transitive （传递）”，它是最难识别和管理的依赖。

如果你将使用 Spring，你需要复制哪些包含你需要的 Spring 功能的 jar包。为了使这个过程更加简单，Spring 被打包为一组模块，这些模块尽可能多的分开依赖关系。例如，如果不想写一个 web 应用程序，你就不需要引入 Spring-web 模块。为了在本指南中标记 Spring 库模块我们使用了速记命名约定 spring- 或者 *spring*-.jar ，其中*代表模块的短名（比如 spring-core,spring-webmvc, spring-jms 等）。实际的jar 文件的名字，通常是用模块名字和版本号级联（如 spring-core-4.3.0.RELEASE.jar）

每个 Spring Framework 发行版本将会放到下面的位置：

- Maven Central （Maven 中央库），这是 Maven 查询的默认库，而不需要任何特殊的配置就能使用。许多常用的 Spring 的依赖库也存在与Maven Central ，并且 Spring 社区的很大一部分都使用 Maven 进行依赖管理，所以这是最方便他们的。jar 命名格式是 `spring-*-<version>.jar` ， Maven groupId 是 `org.springframework`
- 公共 Maven 仓库还拥有 Spring 专有的库。除了最终的 GA 版本，这个库还保存开发的快照和里程碑。JAR 文件的名字是和 Maven Central 相同的形式，所以这是让 Spring 的开发版本使用其它部署在 Maven Central 库的一个有用的地方。该库还包含一个用于发布的 zip 文件包含所有Spring jar，方便下载。

所以首先，你要决定用什么方式管理你的依赖，通常建议你使用自动系统像 Maven, Gradle 或 Ivy, 当然你也可以下载 jar

下面是 Spring 中的组件列表。更多描述，详见2.2. Modules （模块）

Table 2.1. Spring Framework Artifacts

GroupId	ArtifactId	Description
org.springframework	spring-aop	Proxy-based AOP support
org.springframework	spring-aspects	AspectJ based aspects
org.springframework	spring-beans	Beans support, including Groovy
org.springframework	spring-context	Application context runtime, including scheduling and remoting abstractions
org.springframework	spring-context-support	Support classes for integrating common third-party libraries into a Spring application context

org.springframework	spring-core	Core utilities, used by many other Spring modules
org.springframework	spring-expression	Spring Expression Language (SpEL)
org.springframework	spring-instrument	Instrumentation agent for JVM bootstrapping
org.springframework	spring-instrument-tomcat	Instrumentation agent for Tomcat
org.springframework	spring-jdbc	JDBC support package, including DataSource setup and JDBC access support
org.springframework	spring-jms	JMS support package, including helper classes to send and receive JMS messages
org.springframework	spring-messaging	Support for messaging architectures and protocols
org.springframework	spring-orm	Object/Relational Mapping, including JPA and Hibernate support
org.springframework	spring-oxm	Object/XML Mapping
org.springframework	spring-test	Support for unit testing and integration testing Spring components
org.springframework	spring-tx	Transaction infrastructure, including DAO support and JCA integration
org.springframework	spring-web	Web support packages, including client and web remoting
org.springframework	spring-webmvc	REST Web Services and model-view-controller implementation for web applications
org.springframework	spring-webmvc-portlet	MVC implementation to be used in a Portlet environment
org.springframework	spring-websocket	WebSocket and SockJS implementations, including STOMP support

Spring 的依赖以及基于 Spring

虽然 Spring 提供了集成在一个大范围的企业和其他外部工具的支持，它故意保持其强制性依赖关系降到最低：在简单的用例里，你无需查找并下载（甚至自动）一大批 jar 库来使用 Spring 。基本的依赖注入只有一个外部强制性的依赖，这是用来做日志的（见下面更详细地描述日志选项）。

接下来我们将一步步展示如果配置依赖 Spring 的程序，首先用 Maven 然后用 Gradle 和最后用 Ivy。在所有的情况下，如果有不清楚的地方，查看的依赖性管理系统的文档，或看一些示例代码。Spring 本身是使用 Gradle 来管理依赖的，我们的很多示例也是使用 Gradle 或 Maven。

译者注：有关 Gradle 的使用，可以参见笔者的另外一部作品《Gradle 2 用户指南》

Maven 依赖管理

如果您使用的是 Maven 的依赖管理你甚至不需要明确提供日志依赖。例如，要创建一个应用程序的上下文和使用依赖注入来配置应用程序，你的Maven 依赖将看起来像这样：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

就是它。注意 scope 可声明为 runtime （运行时）如果你不需要编译 Spring 的API，这通常是基本的依赖注入使用的案例。

以上与 Maven Central 存储库工程实例。使用 Spring Maven 存储库（如里程碑或开发者快照），你需要在你的 Maven 配置指定的存储位置。如下：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.release</id>
        <url>http://repo.spring.io/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

里程碑：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

以及快照：

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.snapshot</id>
        <url>http://repo.spring.io/snapshot/</url>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>
</repositories>
```

Maven "Bill Of Materials" 依赖

有可能不小心使用不同版本的 Spring JAR 在使用 Maven 时。例如，你可能发现一个第三方的库，或另一 Spring 的项目，拉取了一个在传递依赖较早的发布包。如果你自己忘记了显式声明一个直接依赖，各种意想不到的问题出现。

为了克服这些问题，Maven 支持 "bill of materials" (BOM) 的依赖的概念。你可以在你的 dependencyManagement 部分引入 spring-framework-bom 来确保所有 spring依赖（包括直接和传递的）是同一版本。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>4.3.0.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

使用 BOM 后，当依赖 Spring Framework 组件后，无需指定 `<version>` 属性

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>
<dependencies>
```

Gradle 依赖管理

用 Gradle 来使用 Spring ,在 repositories 中填入适当的 URL :

```
repositories {
    mavenCentral()
    // and optionally...
    maven { url "http://repo.spring.io/release" }
}
```

可以适当修改 URL 从 /release 到 /milestone 或 /snapshot 。库一旦配置，就能声明 Gradle 依赖:

```
dependencies {
    compile("org.springframework:spring-context:4.3.0.RELEASE")
    testCompile("org.springframework:spring-test:4.3.0.RELEASE")
}
```

Ivy 依赖管理

如果你更喜欢用Ivy管理依赖也有类似的配置选项。

配置 Ivy ，将指向 Spring 的库 添加下面的 resolver（解析器）到你ivysettings.xml：

```
<resolvers>
    <ibiblio name="io.spring.repo.maven.release"
            m2compatible="true"
            root="http://repo.spring.io/release/"/>
</resolvers>
```

可以适当修改 root URL 从 /release 到 /milestone 或 /snapshot 。

一旦配置，就能添加依赖，举例 (在 ivy.xml):

```
<dependency org="org.springframework"
    name="spring-core" rev="4.3.0.RELEASE" conf="compile->runtime"/>
```

分发的 zip 文件

虽然使用构建系统，支持依赖管理是推荐的方式获得了 Spring Framework,，它仍然是可下载分发的 zip 文件。

分发的 zip 文件是发布到 Spring Maven Repository （这是为了我们的便利，在下载这些文件的时候你不需要 Maven 或者其他的构建系统）。

下载一个 Zip，在web 浏览器打开 http://repo.spring.io/release/org/springframework/spring ，选择适当的文件夹的版本。下载完毕文件结尾是 -dist.zip，例如， spring-framework-{spring-version}-RELEASE-dist.zip 。分发也支持发布里程碑和快照。

日志

对于 Spring 日志是非常重要的依赖，因为：a）它是唯一的外部强制性的依赖；b）每个人都喜欢从他们使用的工具看到一些输出；c）Spring 结合很多其他工具都选择了日志依赖。应用开发者的一个目标就是往往是有统一的日志配置在一个中心位置为了整个应用程序，包括所有的外部元件。这就更加困难，因为它可能已经有太多选择的日志框架。

在 Spring 强制性的日志依赖 是 Jakarta Commons Logging API（JCL）。我们编译 JCL，我们也使得 JCL `Log` 对象对 Spring Framework 的扩展类可见。所有版本的 Spring 使用同样的日志库，这对于用户来说是很重要的：迁移就会变得容易向后兼容性，即使扩展 Spring的应用程序。我们这样做是为了使 Spring 的模块之一明确依赖 commons-logging (JCL的典型实现)，然后使得其他的所有模块在编译的时候都依赖它。使用 Maven 为例，如果你想知道何处依赖了 commons-logging ，那么就是来自 Spring 的并且明确来自中心模块 spring-core。

关于 commons-logging 的好处是你不需要任何东西就能让你的应用程序程序跑起来。它有一个运行时发现算法，该算法在众所周知的classpath路径下寻找其他的日志框架并且使用它认为适合的（或者你可以告诉它你需要的是哪一个）。如果没有其他的日志框架存在，你可以从JDK（Java.util.logging 或者JUL 的简称）获得日志。在大多数情况下，你可以在控制台查看你的Spring 应用程序工作和日志，并且这是很重要的。

不使用 Commons Logging

不幸的是，commons-logging 的运行时日志框架发现算法，确实方便了最终用户,但却是有问题的。如果我们能够时光倒流，现在从新开始 Spring 项目并且他使用了不同的日志依赖。第一个选择很可能是Simple Logging Facade for Java (SLF4J)，过去也曾被许多其他工具通过 Spring 使用到他们的应用程序。

主要有两种方法可以关闭commons-logging：

1. 通过 spring-core 模块排除依赖（因为它是唯一的显示依赖于 commons-logging 的模块）。
2. 依赖特殊的 commons-logging 依赖，用空的jar（更多的细节可以在SLF4J FAQ中找到）替换掉库。

排除 commons-logging，添加以下的内容到 dependencyManagement 部分：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.0.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

现在，这个应用程序可能运行不了，因为在类路径上没有实现 JCL API，因此要修复它就必须提供一个新的(日志框架)。在下一节我们将向你展示如何提供另一种实现 JCL，使用 SLF4J 作为例子的另一种实现。

使用 SLF4J

SLF4J 是一个更加简洁的依赖，在运行时相对于 commons-logging 更加的有效因为它使用编译时绑定来代替运行时发现其他日志框架的集成。这也意味着，你不得不更加明确你想在运行时发生什么，并相应的声明它或者配置它。SLF4J 提供绑定很多的常见日志框架，因此你可以选择一个你已经使用的，并且绑定到配置和管理。

SLF4J 提供了绑定很多的常见日志框架，包括 JCL，它也做了反向工作:是其他日志框架和它自己之间的桥梁。因此在 Spring 中使用 SLF4J 时，你需要使用 SLF4J-JCL 桥接替换掉 commons-logging 的依赖。一旦你这么做了，Spring 调用日志就会调用 SLF4J API，因此如果在你的应用程序中的其他库使用这个API，那么你就需要有个地方配置和管理日志。

一个常见的选择就是桥接 Spring 和 SLF4J，提供显示的绑定 SLF4J 到Log4J 上。你需要支持 4 个的依赖（排除现有的 commons-logging）：桥接，SLF4J API，绑定 Log4J 和 Log4J 实现自身。在 Maven 中你可以这样做：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.0.RELEASE</version>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

这似乎是一个很大的依赖性，其仅仅是为了得到一些日志文件。它的确如此，但它是可选的，它在关于类加载器的问题上应该比 commons-logging 表现的更加的好，特别是当它运行在在一个严格的容器中像OSGi 平台。据说还有一个性能优势是因为绑定是在编译时而不是在运行时。

对于SLF4J用户来说，一个更常见的选择是，使用更少的步骤和产生更少的依赖，那就是直接绑定 Logback。这消除了多余的绑定步骤，因为 Logback 直接实现了 SLF4J，因此你只需要依赖两个库而不是4个（jcl-over-slf4j 和 logback）。如果你这样做，你可能还需要从其他外部依赖（不是 Spring）排除 slf4j-api 依赖，因为在类路径中你只需要一个版本的API。

使用 Log4J

许多人使用 Log4j 作为日志框架，用于配置和管理的目的。它是有效的和完善的，事实上这也是我们在运行时使用的，当我们构架和测试 Spring 时。Spring 也提供一些配置和初始化 Log4j 的工具，因此在某些模块上它有一个可选的编译时依赖在 Log4j。

为了使 Log4j 工作在默认的 JCL 依赖下（commons-logging），你所需要做的就是把 Log4j 放到类路径下，为它提供配置文件(log4j.properties 或者 log4j.xml 在类路径的根目录下)。因此对于Maven 用户这就是你的依赖声明：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.3.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
    </dependency>
</dependencies>
```

下面是一个 log4j.properties 的实例，用于将日志打印到控制台：

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n

log4j.category.org.springframework.beans.factory=DEBUG
```

Runtime Containers with Native JCL

很多的人在提供 JCL 实现的容器下运行他们的 Spring 应用程序。IBM Websphere Application Server (WAS)为例。这样往往会导致问题，遗憾的是没有一个一劳永逸的解决方案；仅仅简单地排除 commons-logging依赖在大多数情况下是不够的。

要清楚这一点：问题报告通常不是 JCL 本身，或者 commons-logging：相反，他们是绑定 commons-logging 到其他的框架（通常是 Log4j ）。这可能会失败，因为commons-logging 改变了路径，当他们运行时发现在一些容器中找到了老版本（1.0）并且现在大多数人使用的现代版本(1.1)。Spring 不使用 JCL API 任何不常见的模块，所以没有什么问题出现，但是很快 Spring 或者你的应用程序视图做一些日志时，你会发现绑定的 Log4j 不工作了。

在这种情况下，WAS 最容易的是转化类加载器的层次结构（IBM 称之为“parent last”），使应用程序控制 JCL 的依赖关系，而不是容器。该选项不是总是开放的，但是有很多其他的建议在公共领域的替代方法，你的里程可能取决于确切的版本和特定的容器。

在 Spring 4.0 新功能和增强

Spring 框架第一个版本发布于 2004 年，自发布以来已历经三个主要版本更新: Spring 2.0 提供了 XML 命名空间和 AspectJ 支持；Spring 2.5 增加了注释驱动（annotation-driven）的配置支持； Spring 3.0增加了对 Java 5+ 版本的支持和 `@Configuration` 模型。

Spring 4.0 是最新的主要版本，并且首次完全支持 Java 8 的特性。你仍然可以使用老版本的 Java，但是最低版本的要求已经提高到 Java SE 6。我们也借主要版本更新的机会删除了许多过时的类和方法。

你可以在Spring Framework GitHub Wiki上查看 升级 Spring 4.0 的迁移指南 。

改进的入门体验

新的 spring.io 网站提供了一整个系列的 "入门指南" 帮助你学习 Spring。你可以本文档的 1. Getting Started with Spring 一节阅读更多的入门指南。新网站还提供了Spring 之下其他额外项目的一个全面的概述。

如果你是一个 Maven 用户，你可能会对 BOM 这个有用的 POM 文件感兴趣， 这个文件已经与每个 Spring 的发布版发布。

移除不推荐的包和方法

所有过时的包和许多过时的类和方法已经从Spring4中移除。如果你从之前的发布版升级Spring，你需要保证已经修复了所有使用过时的API方法。

查看完整的变化： API差异报告。

请注意，所有可选的第三方依赖都已经升级到了最低2010/2011(例如Spring4 通常只支持2010 年的最新或者现在的最新发布版本):尤其是 Hibernate 3.6+、EhCache 2.1+、Quartz 1.8+、Groovy 1.8+、Joda-Time 2.0+。但是有一个例外，Spring4依赖最近的Hibernate Validator 4.3+，现在对Jackson的支持集中在2.0+版本 (Spring3.2支持的Jackson 1.8/1.9，现在已经过时）。

Java 8(以及6和7)

Spring4 支持 Java8 的一些特性。你可以在 Spring 的回调接口中使用 lambda 表达式 和 方法引用。支持 `java.time` (JSR-310)的值类型和一些改进过的注解，例如 `@Repeatable` 。你还可以使用 Java8 的参数名称发现机制（基于 `-parameters` 编译器标志）。

Spring 仍然兼容老版本的 Java 和 JDK：Java SE 6（具体来说，支持JDK6 update 18 ）以上版本，我们建议新的基于 Spring4 的项目使用Java7或Java8。

Java EE 6 和 7

Java EE 6 或以上版本是 Spring4 的底线,与 JPA2.0 和 Servlet3.0规范有着特殊的意义。为了保持与 Google App Engine 和旧的应用程序服务器兼容,Spring4 可以部署在 Servlet2.5 运行环境。但是我们强烈的建议您在 Spring 测试和模拟测试的开发环境中使用 Servlet3.0+。

如果你是*WebSphere 7*的用户，一定要安装*JPA2.0*功能包。在*WebLogic 10.3.4*或更高版本，安装附带的*JPA2.0*补丁。这样就可以将这两种服务器变成*Spring4*兼容的部署环境。

从长远的观点来看，Spring4.0 现在支持 Java EE 7 级别的适用性规范：尤其是 JMS 2.0, JTA 1.2, JPA 2.1, Bean Validation 1.1 和JSR-236 并发工具类。像往常一样，支持的重点是独立的使用这些规范。例如在 Tomcat 或者独立环境中。但是，当把 Spring 应用部署到 Java EE 7 服务器时它同样适用。

注意，Hibernate 4.3 是 JPA 2.1 的提供者，因此它只支持 Spring4。同样适用用于作为 Bean Validation 1.1 提供者的 Hibernate Validator 5.0。这两个都不支持 Spring3.2。

Groovy Bean Definition DSL

Spring4.0 支持使用 Groovy DSL 来进行外部的 bean 定义配置。这在概念上类似于使用 XML 的 bean 定义，但是支持更简洁的语法。使用Groovy 还允许您轻松地将 bean 定义直接嵌入到引导代码中。例如：

```
def reader = new GroovyBeanDefinitionReader(myApplicationContext)
reader.beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

有关更多信息，请参阅 `GroovyBeanDefinitionReader` javadocs

核心容器改进

有几处对核心容器的常规改进：

- Spring 现在注入 Bean 的时候把 泛型类型当成一种形式的限定符。例如：如果你使用Spring Data `Repository` 你可以方便的插入特定的实现：`@Autowired Repository<Customer> customerRepository` 。
- 如果你使用 Spring 的元注解支持，你现在可以开发自定义注解来公开源注解的特定属性。
- 当自动装配到lists和arrays时，Beans 现在可以进行排序 了。支持 `@Order` 注解和 `Ordered` 接口两种方式。 `@Lazy` 注解现在可以用在注入点以及 `@Bean` 定义上。
- 引入 `@Description` 注解,开发人员可以使用基于Java 方式的配置。
- 根据条件筛选 Beans的广义模型通过 `@Conditional` 注解加入。这和 `@Profile` 支持的类似，但是允许以编程式开发用户定义的策略。
- 基于CGLIB的代理类不在需要默认的构造方法。这个支持是由 objenesis库提供。这个库重新打包到 Spring 框架中，作为Spring框架的一部分发布。通过这个策略，针对代理实例被调用没有构造可言了。
- 框架现在支持管理时区。例如 `LocaleContext` 。

常规Web改进

现在仍然可以部署到 Servlet 2.5 服务器，但是 Spring4.0 现在主要集中在 Servlet 3.0+ 环境。如果你使用 Spring MVC 测试框架，你需要将 Servlet 3.0 兼容的 JAR 包放到 测试的 *classpath* 下。

除了稍后会提到的 WebSocket 支持外，下面的常规改进已经加入到Spring 的 Web 模块：

- 你可以在Spring MVC应用中使用新的 `@RestController` 注解，不在需要给 `@RequestMapping` 的方法添加 `@ResponseBody` 注解。
- AsyncRestTemplate 类已被添加进来，当开发 REST 客户端时，允许非阻塞异步支持。
- 当开发 Spring MVC 应用时，Spring现在提供了全面的时区支持 。

WebSocket, SockJS, 和 STOMP 消息

一个新的 `spring-websocket` 模块提供了全面的基于 WebSocket 和在Web 应用的客户端和服务器之间双向通信的支持。它和 Java WebSocket API JSR-356 兼容，此外还提供了当浏览器不支持 WebSocket 协议时 (如 Internet Explorer < 10)的基于SockJS 的备用选项(如 WebSocket emulation)。

一个新的 `spring-messaging` 模块添加了支持 STOMP 作为 WebSocket 子协议用于在应用中使用注解编程模型路由和处理从 WebSocket 客户端发送的 STOMP 消息。因此 `@Controller` 现在可以同时包含 `@RequestMapping` 和 `@MessageMapping` 方法用于处理 HTTP 请求和来自 WebSocket 连接客户端发送的消息。新的 `spring-messaging` 模块还包含了来自以前 Spring 集成项目的关键抽象，例如 `Message` 、 `MessageChannel` 、 `MessageHandler` 等等, 以此作为基于消息传递的应用程序的基础。

欲知详情以及较全面的介绍，请参见Chapter 20, WebSocket 支持一节。

测试改进

除了精简 `spring-test` 模块中过时的代码外，Spring 4 还引入了几个用于单元测试和集成测试的新功能。

- 几乎 `spring-test` 模块中所有的注解（例如：`@ContextConfiguration` 、`@WebAppConfiguration` 、`@ContextHierarchy` 、`@ActiveProfiles` 等等)现在可以用作元注解来创建自定义的 *composed annotations* 并且可以减少测试套件的配置。
- 现在可以以编程方式解决Bean定义配置文件的激活。只需要实现一个自定义的ActiveProfilesResolver，并且通过@ActiveProfiles的resolver属性注册。
- 新的 `SocketUtils` 类被引入到了 `spring-core` 模块。这个类可以使你能够扫描本地主机的空闲的 TCP 和 UDP 服务端口。这个功能不是专门用在测试的，但是可以证明在你使用 Socket 写集成测试的时候非常有用。例如测试内存中启动的SMTP服务器，FTP服务器，Servlet容器等。
- 从 Spring 4.0 开始, `org.springframework.mock.web` 包中的一套mock是基于Servlet 3.0 API。此外，一些Servlet API mocks（例如：`MockHttpServletRequest` 、`MockServletContext` 等等）已经有一些小的改进更新，提高了可配置性。

JMS 改进

Spring 4.1 引入了一个更简单的基础架构，使用 `@JmsListener` 注解bean 方法来注册 JMS 监听端点。XML 命名空间已经通过增强来支持这种新的方式（ `jms:annotation-driven` ），它也可以完全通过Java配置(`@EnableJms` , `JmsListenerContainerFactory`)来配置架构。也可以使用 `JmsListenerConfigurer` 注解来注册监听端点。

Spring 4.1 还调整了 JMS 的支持，使得你可以从 `spring-messaging` 在 Spring4.0 引入的抽象获益，即：

- 消息监听端点可以有更为灵活的签名，并且可以从标准的消息注解获益，例如 `@Payload` 、 `@Header` 、 `@Headers` 和@ `SendTo` 注解。另外，也可以使用一个标准的消息，以代替 `javax.jms.Message` 作为方法参数。
- 一个新的可用 `JmsMessageOperations` 接口和允许操作使用 `Message` 抽象的 `JmsTemplate` 。

最后，Spring 4.1提供了其他各种各样的改进：

- JmsTemplate中的同步请求-答复操作支持
- 监听器的优先权可以指定每个 `<jms:listener/>` 元素
- 消息侦听器容器恢复选项可以通过使用 `BackOff` 实现进行配置
- JMS 2.0消费者支持共享

缓存改进

Spring 4.1 支持JCache (JSR-107)注解使用Spring的现有缓存配置和基础结构的抽象；使用标准注解不需要任何更改。

Spring 4.1也大大提高了自己的缓存抽象：

- 缓存可以在运行时使用 `CacheResolver` 解决。因此使用 `value` 参数定义的缓存名称不在是强制性的。
- 更多的操作级自定义项：缓存解析器，缓存管理器，键值生成器
- 一个新的 `@CacheConfig` 类级别注解允许在类级别上共享常用配置，不需要启用任何缓存操作。
- 使用 `CacheErrorHandler` 更好的处理缓存方法的异常

Spring 4.1为了在 `CacheInterface` 添加一个新的 `putIfAbsent` 方法也做了重大的更改。

Web 改进

- 现有的基于 `ResourceHttpRequestHandler` 的资源处理已经扩展了新的抽象 `ResourceResolver` ， `ResourceTransformer` 和 `ResourceUrlProvider` 。一些内置的实现提供了版本控制资源的 URL（有效的 HTTP 缓存），定位 gzip 压缩的资源，产生 HTML5 AppCache清单，以及更多的支持。参见第17.16.7，“Serving of Resources(服务资源)”。
- JDK 1.8 的 `java.util.Optional` 现在支持 `@RequestParam` ， `@RequestHeader` 和 `@MatrixVariable` 控制器方法的参数。
- `ListenableFuture` 支持作为返回值替代 `DeferredResult` 所在的底层服务（或者调用 `AsyncRestTemplate` ）已经返回 `ListenableFuture` 。
- `@ModelAttribute` 方法现在依照相互依存关系的顺序调用。见SPR-6299。
- Jackson的 `@JsonView` 被直接支撑在 `@ResponseBody` 和 `ResponseEntity` 控制器方法用于序列化不同的细节对于相同的 POJO（如摘要与细节页）。同时通过添加序列化视图类型作为模型属性的特殊键来支持基于视图的渲染。见Jackson Serialization View Support(Jackson序列化视图支持)
- Jackson 现在支持 JSONP ，见 Jackson JSONP Support
- 一个新的生命周期选项可用于在控制方法返回后，响应被写入之前拦截 `@ResponseBody` 和 `ResponseEntity` 方法。要充分利用声明 `@ControllerAdvice` bean 实现 `ResponseBodyAdvice` 。为 `@JsonView` 和 JSONP 的内置支持利用这一优势。参见第17.4.1，“使用HandlerInterceptor 拦截请求”。
- 有三个新的 `HttpMessageConverter` 选项：
    - GSON - 比 Jackson 更轻量级的封装;已经被使用在 Spring Android
    - Google Protocol Buffers - 高效和有效的企业内部跨业务的数据通信协议，但也可以用于浏览器的 JSON 和 XML 的扩展
    - Jackson 基于 XML 序列化，现在通过 jackson-dataformat-xml 扩展得到了支持。如果 `jackson-dataformat-xml` 在 classpath， 默认情况下使用 `@EnableWebMvc` 或 `<mvc:annotation-driven/>` ，这是，而不是JAXB2。
- 如 JSP 等视图现在可以通过名称参照控制器映射建立链接控制器。默认名称分配给每一个 `@RequestMapping` 。例如 `FooController` 的方法与 `handleFoo` 被命名为“FC#handleFoo”。命名策略是可插拔的。另外，也可以通过其名称属性明确命名的 `@RequestMapping` 。在Spring JSP标签库的新 `mvcUrl` 功能使这个简单的JSP页面中使用。参见第17.7.2，“Building URIs to Controllers and methods from views”
- `ResponseEntity` 提供了一种 builder 风格的 API 来指导控制器向服务器端的响应的展示，例如，ResponseEntity.ok()。
- `RequestEntity` 是一种新型的，提供了一个 builder 风格的 API 来引导客户端的 REST 响应 HTTP 请求的展示。
- MVC 的 Java 配置和 XML 命名空间：

  - 视图解析器现在可以配置包括内容协商的支持，请参见17.16.6“视图解析”。
  - 视图控制器现在已经内置支持重定向和设置响应状态。应用程序可以使用它来配置重定向的URL，404 视图的响应，发送“no content”的响应，等等。有些用例这里列出。
  - 现在内置路径匹配的自定义。参见第17.16.9，“路径匹配”。
- Groovy 的标记模板支持（基于Groovy的2.3）。见 `GroovyMarkupConfigurer` 和各自的 `ViewResolver` 和“视图”的实现。

WebSocket STOMP 消息 改进

- SockJS(Java)客户端支持.查看 `SockJsClient` 和在相同包下的其他类.
- 发布新的应用上下文实践 `SessionSubscribeEvent` 和 `SessionUnsubscribeEvent` ,用于STOMP客户端的订阅和取消订阅.
- 新的"websocket"级别.查看 Section 25.4.14, “WebSocket Scope”.
- `@SendToUser` 仅仅靠一个会话就可以命中,而不一定需要一个授权的用户.
- `@MessageMapping` 方法使用 **.** 来代替 **/** 作为目录分隔符。查看 SPR-11660.。
- STOMP/WebSocket 监听消息的收集和记录。查看25.4.16, “Runtime Monitoring”.。
- 显著优化和改善在调试模式下依然保持日志的可读性和简洁性。
- 优化消息创建，包含对临时消息可变性的支持和避免自动消息ID和时间戳的创建。查看 `MessageHeaderAccessor` 的java文档。
- 在WebSocket会话建立之后的1分钟没有任何活动,则关闭STOMP/WebSocket连接。

测试 改进

- Groovy脚本现在能使用 `ApplicationContext` 配置，在TestContext框架中整合测试。the section called “Context configuration with Groovy scripts”
- 现在通过新的 `TestTransaction` 接口，使用编程化来开始结束测试管理事务。the section called “Programmatic transaction management”
- 现在SQL脚本的执行可以通过 `Sql` 和 `SqlConfig` 注解申明在每一个类和方法中。the section called “Executing SQL scripts”
- 测试属性值可以通过配置 `@TestPropertySource` 注解来自动覆盖系统和应用的属性值。the section called “Context configuration with test property sources”。
- 默认的 `TestExecutionListeners` 现在可以自动被发现。the section called “Automatic discovery of default TestExecutionListeners” 。
- 自定义的 `TestExecutionListeners` 现在可以通过默认的监听器自动合并。the section called “Merging TestExecutionListeners”。
- 在TestContext框架中的测试事务方面的文档已经通过更多解释和其他事例得到改善。the section called “Transaction management”。
- `MockServletContext` 、 `MockHttpServletRequest` 和其他servlet接口mocks等诸多改善。
- `AssertThrows` 重构后， `Throwable` 代替 `Exception` 。
- Spring MVC测试中，使用JSONPath选项返回JSON格式可以使用JSON Assert来断言,非常像之前的XML和XMLUnit。
- `MockMvcBuilder` 现在可以在 `MockMvcConfigurer` 的帮助下创建。 `MockMvcConfigurer` 的加入使得Spring Security的设置更加简单，同时使用于任何第三方的普通封装设置或则仅仅在本项目中。
- `MockRestServiceServer` 现在支持 `AsyncRestTemplate` 作为客户端测试。
- 发布新的应用上下文实践 `SessionSubscribeEvent` 和 `SessionUnsubscribeEvent` ,用于STOMP客户端的订阅和取消订阅.
- 新的"websocket"级别.查看[Section 21.4.13, “WebSocket Scope”.](http://docs.spring.io/spring/docs/current/spring-framework-

核心容器改进

- 如 @bean 注释,就如同得到发现和处理 Java 8 默认方法一样,可以允许组合配置类与默认 @bean 接口方法。
- 配置类现在可以声明 @import 作为常规组件类,允许引入的配置类和组件类进行混合。
- 配置类可以声明一个 @Order 值,用来得到相应的处理顺序(例如重写 bean 的名字)，即使通过类路径扫描检测。
- @Resource 注入点支持 @Lazy 声明,类似于 @autowired, 用于接收用于请求目标 bean 的懒初始化代理。
- 现在的应用程序事件基础架构提供了一个基于注解的模型以及发布任意事件的能力。
  - 任何受管 bean 的公共方法使用 @EventListener 注解来消费事件。
  - @TransactionalEventListener 提供事务绑定事件支持。
- Spring Framework 4.2引入了一流的支持声明和查找注释属性的别名。新 @AliasFor 注解可用于声明一双别名属性在一个注释中或从一个属性在一个声明一个别名定义注解在元注释一个属性组成。
  - 下面的注解已加了 @AliasFor 为了支持提供更有意义的 value 属性的别名: @Cacheable, @CacheEvict, @CachePut, @ComponentScan, @ComponentScan.Filter, @ImportResource, @Scope, @ManagedResource, @Header, @Payload, @SendToUser, @ActiveProfiles, @ContextConfiguration, @Sql, @TestExecutionListeners, @TestPropertySource, @Transactional, @ControllerAdvice, @CookieValue, @CrossOrigin, @MatrixVariable, @RequestHeader, @RequestMapping, @RequestParam, @RequestPart, @ResponseStatus, @SessionAttributes, @ActionMapping, @RenderMapping, @EventListener, @TransactionalEventListener
  - 例如，spring-test 的 @ContextConfiguration 现在声明如下：

```
public @interface ContextConfiguration {

    @AliasFor("locations")
    String[] value() default {};

    @AliasFor("value")
    String[] locations() default {};

    // ...
}
```

```
* 同样, 组合注解（composed annotations）从元注解覆盖的属性,现在可以使用 @AliasFor 进行细粒度控制哪些属性是覆盖在一个注释的层次结构。事实上,现在可以声明一个别名给元注释的 value 属性。
* 例如，开发一个组合注解用于一个自定义的属性的覆盖
```

```
@ContextConfiguration
public @interface MyTestConfig {

    @AliasFor(annotation = ContextConfiguration.class, attribute = "value")
    String[] xmlFiles();

    // ...
}
```

```
* 见 [Spring Annotation Programming Model](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#annotation-programming-model)
```

- 许多改进Spring的搜索算法用于寻找元注解。例如,局部声明组合注解现在喜欢继承注解。
- 从元注解覆盖属性的组合注解，可以被发现在接口和 abstract, bridge, & interface 方法就像在类，标准方法，构造函数，和字段。
- Map 表示的注解属性(和 AnnotationAttributes 实例)可以 synthesized (合成，即转换)成一个注解。
- 基于字段的数据绑定的特点(DirectFieldAccessor)与当前的基于属性的数据绑定关联(BeanWrapper)。特别是,基于字段的绑定现在支持集合,数组和 Map 的导航。
- DefaultConversionService 现在提供开箱即用的转化器给 Stream, Charset, Currency, 和 TimeZone. 这些转换器可以独立的添加到任何 ConversionService
- DefaultFormattingConversionService 提供开箱即用的支持 JSR-354 的 Money & Currency 类型 (前提是 'javax.money' API 出现在 classpath): 这些被命名为 MonetaryAmount 和 CurrencyUnit。支持使用 @NumberFormat
- @NumberFormat 现在作为元注解使用
- JavaMailSenderImpl 中新的 testConnection() 方法用于检查与服务器的连接
- ScheduledTaskRegistrar 用于暴露调度的任务
- Apache commons-pool2 现在支持用于 AOP CommonsPool2TargetSource 的池化
- 引入 StandardScriptFactory 作为脚本化 bean 的 JSR-223 的基本机制，通过 XML 中的 `lang:std` 元素暴露。支持如 JavaScript 和 JRuby。（注意：JRubyScriptFactory 和 `lang:jruby` 现在不推荐使用了，推荐用 JSR-223）

数据访问改进

- javax.transaction.Transactional 现在可以通过 AspectJ 支持
- SimpleJdbcCallOperations 现在支持命名绑定
- 完全支持 Hibernate ORM 5.0: 作为 JPA 供应商 (自动适配)和原生的 API 一样 (在新的 org.springframework.orm.hibernate5 包中涵盖了该内容)
- 嵌入式数据库可以自动关联唯一名字，并且 `<jdbc:embedded-database>` 支持新的 database-name 属性。见下面“测试改进”内容

JMS 改进

- autoStartup 属性可以通过 JmsListenerContainerFactory 进行控制
- 应答类型 Destination 可以配置在每个监听器容器器
- @SendTo 的值可以用 SpEL 表达式
- 响应目的地可以通过 JmsResponse 在运行时计算
- @JmsListener 是可以可以重复的注解用于声明多个 JMS 容器在相同的方法上 (若你还没有用上 Java8 请使用新引入的 @JmsListeners)。

Web 改进

- 支持 HTTP Streaming 和 Server-Sent Events , se见 “HTTP Streaming”
- 内建支持 CORS ，包括全局 (MVC Java 配置和 XML 命名空间) 和本地 (如 @CrossOrigin) 配置。见 26 章, CORS 支持
- HTTP 缓存升级
    - 新的 CacheControl 构建器; 插入 ResponseEntity, WebContentGenerator, ResourceHttpRequestHandler
    - 改进的 ETag/Last-Modified 在 WebRequest 中支持
- 自定义映射注解使用 @RequestMapping 作为 元数据注解
- AbstractHandlerMethodMapping 中的 public 方法用于运行时注册和注销请求映射
- AbstractDispatcherServletInitializer 中的 Protected createDispatcherServlet 方法用来进一步自定义 DispatcherServlet 实例
- HandlerMethod 作为 @ExceptionHandler方法的方法参数,特别是方便 @ControllerAdvice 组件
- java.util.concurrent.CompletableFuture 作为 @Controller 方法返回值类型
- 字节范围（Byte-range）的请求支持在 HttpHeaders，用于静态资源
- @ResponseStatus 发现嵌套异常。
- 在 RestTemplate 中的 UriTemplateHandler 扩展端点
    - DefaultUriTemplateHandler 暴露 baseUrl 属性和路径段的编码选项
    - 扩展端点可以使用插入任何 URI 模板库
- OkHTTP 与 RestTemplate 集成
- 自定义 baseUrl 在 MvcUriComponentsBuilder 选择方法。
- 序列化/反序列化异常消息现在记录为 WARN 级别
- 默认的 JSON 前缀改变了从 `{}&&` 改为更安全的 `)]}',`
- 新的 RequestBodyAdvice 扩展点和内置的实现支持 Jackson 的 在 @RequestBody 的 @JsonView
- 当使用 GSON 或 Jackson 2.6 +,处理程序方法的返回类型是用于提高参数化类型的序列化，比如 `List<Foo>`
- 引入的 ScriptTemplateView 作为 JSR-223 的脚本化 web 视图机制为基础,关注 JavaScript 视图模板 Nashorn (JDK 8)。

WebSocket 消息改进

- 暴露展示信息关于用户的连接和订阅:
  - 新 SimpUserRegistry 公开为一个名为“userRegistry”的bean。
  - 共享在服务器集群的展示信息(见代理中继配置选项)
- 解决用户目的地在集群的服务器(见代理中继配置选项)。
- StompSubProtocolErrorHandler 扩展端点用来自定义和控制 STOMP ERROR 帧给用户
- 全局 @MessageExceptionHandler 方法通过 @ControllerAdvice 组件
- 心跳和 SpEL 表达式'selector'头用 SimpleBrokerMessageHandler 订阅
- STOMP 客户端使用TCP 和 WebSocket; 见 25.4.13, “STOMP 客户端”
- @SendTo 和 @SendToUser 可以包含目标变量的占位符。 Jackson 的 @JsonView 支持 @MessageMapping 和 @SubscribeMapping 方法返回值
- ListenableFuture 和 CompletableFuture 是从 @MessageMapping 和 @SubscribeMapping 方法返回类型值
- MarshallingMessageConverter 用于 XML 有效载荷

测试改进

- 基于 JUnit 集成测试现在可以执行 JUnit 规则而不是SpringJUnit4ClassRunner。这允许基于 spring 的集成测试与运行JUnit 的 Parameterized 或第三方 运行器 MockitoJUnitRunner 等。详见 Spring JUnit 规则
- Spring MVC Test 框架，现在支持第一类 HtmlUnit，包括集成 Selenium's WebDriver,允许基于页面的 Web 应用测试而无需部署到 Servlet 容器。详见 14.6.2, "HtmlUnit 集成"
- AopTestUtils 是一个新的测试工具，允许开发者获得潜在的目标对象的引用隐藏在一个或多个 Spring 代理。详见 13.2.1, "常见测试工具"
- ReflectionTestUtils 现在支持 setting 和 getting static 字段,包括常量
- bean 定义归档文件的原始顺序，通过 @ActiveProfiles 声明，现在保留为了支持用例,如 Spring 的 ConfigFileApplicationListener 引导加载配置文件基于活动归档文件的名称。
- @DirtiesContext 支持新 BEFORE_METHOD BEFORE_CLASS,BEFORE_EACH_TEST_METHOD 模式，用于测试之前关闭 ApplicationContext——例如,如果一些烦人的(即，有待确定)测试在一个大型测试套件的 ApplicationContext 的原始配置已经损坏。
- @Commit 是新的注解直接可以用来代替 @Rollback(false)
- @Rollback 用来配置类级别的默认回滚语义
  - 因此,现在 @TransactionConfiguration 弃用,在后续版本将被删除。
- @Sql 现在支持内联 SQL 语句的执行通过一个新的 statements 属性
- ContextCache 用于缓存测试之间的 ApplicationContext，而现在这是一个公开的 API ，默认的实现可以替代自定义的缓存需求
- DefaultTestContext, DefaultBootstrapContext, 和 DefaultCacheAwareContextLoaderDelegate 现在是公开的类，支持子包，允许自定义扩展
- TestContextBootstrapper 现在负责构建 TestContext
- 在 Spring MVC Test 框架，MvcResult 详情可以被日志记录在 DEBUG 级别或者写入自定义的 OutputStream 或 Writer。详见 log(), print(OutputStream), 和 MockMvcResultHandlers 的 print(Writer) 方法
- JDBC XML名称空间支持一个新的 `<jdbc:embedded-database>` 的 database-name 属性,允许开发人员为嵌入式数据库设置独特的名字——例如,通过一个 SpEL 表达式或 前活动bean定义配置文件所影响的占位符属性
- 嵌入式数据库现在可以自动分配一个唯一的名称,允许常用的测试数据库配置在不同的 ApplicationContext 的测试套件中。 参见18.8.6"给嵌入式数据库生成惟一名称"的细节。

Spring 4.3 的新功能和增强

核心容器改进

- 核心容器额外提供了更丰富的元数据来改进编程。
- 默认 Java 8 的方法检测为 bean 属性的 getter/setter 方法。
- 如果目标 bean 只定义了一个构造函数,则它无需要指定 `@Autowired` 注解
- `@Configuration` 类支持构造函数注入。
- 任何 SpEL 表达式用于指定 `@EventListener` 的 `condition` 引用到 bean（例如 `@beanName.method()` ）。
- 组成注解现在可以用一个包含元注解中的数组属性的数组组件类型的元素来覆盖。例如， `@RequestMapping` 的的 `String[] path` 可以在组成注解用 `String path` 覆盖。
- `@Scheduled` 和 `@Schedules` 现在是作为元注解用来通过属性覆盖来创建自定义的组成注解。
- `@Scheduled` 适当支持任何范围内的 bean。

数据访问改进

`jdbc:initialize-database` 和 `jdbc:embedded-database` 支持可配置的分离器被应用到每个脚本。

缓存改进

Spring 4.3 允许在一个给定的 key 并发调用时实现要同步，使得相应的值只计算一次。这是一个可选的功能，通过设置 `@Cacheable` 的新的 `sync` 属性来启用。此功能引入了 `Cache` 接口的一个重大更改，即 `get(Object key, Callable<T> valueLoader)` 方法已添加。

Spring 4.3 还改进了缓存抽象如下：

- SpEL 表达式对于缓存相关的注解，现在可以引用 bean（即 `@beanName.method())` ）。
- `ConcurrentMapCacheManager` 和 `ConcurrentMapCache` 现在通过一个新的 `storeByValue` 属性支持缓存实体的序列化。 `@Cacheable` ， `@CacheEvict` ， `@CachePut` 和 `@Caching` 现在是作为元注解用来通过属性覆盖来创建自定义的组成注解。

JMS 改进

- `@SendTo` 现在可以在类级别指定一个共同回复目标。
- `@JmsListener` 和 `@JmsListeners` 现在是作为元注解用来通过属性覆盖来创建自定义的组成注解。

Web 改进

- 内建支持 HTTP HEAD 和 HTTP OPTIONS.
- 新的组合注解 `@GetMapping` , `@PostMapping` , `@PutMapping` , `@DeleteMapping` , 和 `@PatchMapping` 用于 `@RequestMapping` 。
  - 详见 `@RequestMapping` 组合变种
- 新的 `@RequestScope` , `@SessionScope` , 和 `@ApplicationScope` 用于 web 范围的组合注解
  - Request scope, Session scope, 和 Application scope
- 新的 `@RestControllerAdvice` 注解是 `@ControllerAdvice` 和 `@ResponseBody` 的语义结合
- `@ResponseStatus` 现在在类级别被支持，并被所有方法继承
- 新的 `@SessionAttribute` 注解用于访问 session 属性 (见例子)
- 新的 `@RequestAttribute` 注解用于访问请求属性 (见例子)
- `@ModelAttribute` 允许通过 `binding=false` 来避免数据绑定(见引用)
- 错误和自定义抛出，将被统一到 MVC 异常处理器中处理
- HTTP 消息转换编码一致处理，包括默认 UTF-8 用于多部分文本内容
- 静态资源处理使用配置的 `ContentNegotiationManager` 用于媒体类型计算
- `RestTemplate` 和 `AsyncRestTemplate` 支持通过 `DefaultUriTemplateHandler` 来实现严格的URI变量编码
- `AsyncRestTemplate` 支持请求拦截

WebSocket 消息改进

`@SendTo` 和 `@SendToUser` 现在可以在类级被指定为共享共同的目的地。

测试改进

- 为了支持 Spring TestContext Framework ，现在需要 JUnit 4.12 或者更高的版本
- 新的 `SpringRunner` 关联于 `SpringJUnit4ClassRunner`
- 测试相关的注解，现在可以在接口上声明了。例如，基于 Java 8 的接口上使用测试接口
- 空声明的 `@ContextConfiguration` 现在将会完全忽略，如果检测到默认的 XML 文件, Groovy 脚本, 或 `@Configuration` 类型
- `@Transactional` 测试方法不再需要 `public` (如, 在 TestNG 和 JUnit 5)
- `@BeforeTransaction` 和 `@AfterTransaction` 不再需要 `public` ，并且在 基于 Java 8 的接口的默认方法上声明

- 在Spring TestContext Framework 的 `ApplicationContext` 的缓存现在有界为32默认最大规模和最近最少使用驱逐策略。最大的大小可以通过设置称为 `spring.test.context.cache.maxSize` 一个 JVM 系统属性或 Spring 配置。
- `ContextCustomizer` API 用于自定义测试 `ApplicationContext` 在 bean 定义加载到上下文后但在上下文被刷新前。定制工具可以在全球范围由第三方进行注册，而无需要实现一个自定义的 `ContextLoader` 。
- `@Sql` 和 `@SqlGroup` 现在作为元注解通过覆盖属性来创建自定义组合注解
- `ReflectionTestUtils` 现在在 set 或 get 一个字段时，会自动解开代理。
- 服务器端的 Spring MVC 测试支持具有多个值的响应头。
- 服务器端的 Spring MVC 测试解析表单数据的请求内容和填充请求参数。
- 服务器端的 Spring MVC 测试支持 mock 式的断言来调用处理程序方法。
- 客户端 REST 测试支持允许指定多少次预期的请求以及期望的声明顺序是否应该被忽略（参见15.6.3，“客户端REST测试”）。
- 客户端 REST 测试支持请求主体表单数据的预期。

支持新的类库和服务器

- Hibernate ORM 5.2 (同样很好的支持 4.2/4.3 和 5.0/5.1，不推荐 3.6)
- Jackson 2.8 (在Spring 4.3，最低至 Jackson 2.6+)
- OkHttp 3.x (仍然并行支持 OkHttp 2.x)
- Netty 4.1
- Undertow 1.4
- Tomcat 8.5.2 以及 9.0 M6

Part III. 核心技术

该部分的参考文档涵盖了Spring Framework中所有绝对不可或缺的技术。

这其中最重要的部分就是Spring Framework中的控制反转（IoC)容器。Spring Framework中IoC容器是紧随着Spring中面向切面编程（AOP)技术的全面应用的来完整实现的。Spring Framework有它自己的一套AOP框架，这套框架从概念上很容易理解，而且成功解决了Java企业级应用中AOP需求80%的核心要素。

同样Spring与AspectJ的集成（目前从功能上来说是最丰富，而且也无疑是Java企业领域最成熟的AOP实现）也涵盖在内。

介绍 Spring IoC 容器和 bean

本章涵盖了 Spring Framework 中控制反转原则（IoC）的实现。IoC 也就是被大家所熟知的依赖注入（DI）。他是一个对象定义他的依赖的一个过程，所谓依赖，就是和它一起工作的对象，这个过程只能通过构造函数的参数，工厂方法的参数，或者已经被构造或者从工厂方法返回的对象的 setter 方法设置其属性来实现。接着容器在创建Bean后注入这些依赖。这个过程从根本上来讲是反向的，由于bean自己控制实例，或者直接通过类的结构，类似于 Service Locator 模式来定位他的依赖。

org.springframework.beans 和 org.springframework.context 包是 Spring Framework 的 IoC 容器的根本。BeanFactory 接口提供了一种更先进的配置机制来管理任意类型的对象。ApplicationContext 是 BeanFactory 的一个子接口。ApplicationContext 使得和 Spring 的 AOP 功能集成变得更简单；添加了信息资源处理（国际化中使用），事件发布；还添加了应用程序层的特殊上下文 ，如用于 web 应用程序的 WebApplicationContext。

简而言之，BeanFactory提供了配置框架和基本功能，而ApplicationContext 添加了更多企业应用功能。ApplicationContext完整扩展了BeanFactory, 这些内容在介绍Spring的IoC容器的章节里面会专门讲到。更多使用 BeanFactory 请参阅 章节 6.16, “The BeanFactory”.

在Spring中，由Spring IoC容器管理的，构成你的程序骨架的这些对象叫做bean。 bean对象是指经过IoC容器实例化，组装和管理的对象。此外，bean 就是你应用程序中诸多对象之一。bean 和 bean 的依赖被容器所使用的配置元数据所反射。

容器总览

org.springframework.context.ApplicationContext 代表 Spring IoC 容器，并负责实例化，配置和组装上述 bean 的接口。容器是通过对象实例化，配置，和读取配置元数据汇编得到对象的构建。配置元数据可以是用 XML，Java 注解，或 Java 代码来展示。它可以让你描述组成应用程序的对象和对象间丰富的相互依赖。

Spring ApplicationContext 接口提供了几种即装即用的实现方式。在独立应用中，通常以创建 ClassPathXmlApplicationContext 或FileSystemXmlApplicationContext 的实例。虽然 XML 一直是传统的格式来定义配置元数据，但也可以指示容器使用 Java 注解或代码作为元数据格式，并通过提供少量的XML配置以声明方式启用这些额外的元数据格式的支持。

在大多数应用场合，不需要明确的用户代码来实例化一个 Spring IoC 容器的一个或多个实例。例如，在 web 应用程序中，在应用程序的 web.xml文件中一个简单的样板网站的 XML 描述符通常就足够了（见第6.15.4，“便捷的 ApplicationContext 实例化 Web 应用程序”）。如果您使用的是基于Eclipse的 Spring Tool Suite开发环境，该样板配置可以很容易地用点击几下鼠标或键盘创建。

下面的图展示了 Spring 是如何工作的高级视图。您的应用程序的类与配置元数据进行结合，以便在 ApplicationContext创建和初始化后，你有一个完全配置和可执行的系统或应用程序。

Figure 6.1. The Spring IoC container



配置元数据

如上述图所示，Spring IoC 容器使用 配置元数据（ configuration metadata）;这个配置元数据代表了应用程序开发人员告诉 Spring 容器在应用程序中如何来实例化，配置和组装对象。

传统的配置元数据是一个简单而直观的 XML 格式，这是大多数本章用来传达关键概念和 Spring IoC 容器的功能。

基于 *XML* 的元数据并不是配置元数据的唯一允许的形式。 *Spring IoC*容器本身是完全与配置元数据的实际写入格式分离的。现在，许多开发人员选择基于 *Java* 的配置。来开发他们的应用程序

更多其他格式的元数据见：

- 基于注解的配置：Spring 2.5 的推出了基于注解的配置元数据支持。
- 基于Java的配置：Spring3.0 开始，由 Spring JavaConfig 项目提供了很多功能成为核心 Spring 框架的一部分。因此，你可以通过使用Java，而不是 XML 文件中定义外部 bean 到你的应用程序类。要使用这些新功能，请参阅 @Configuration，@Bean，@Import 和

@DependsOn 注解。

Spring 配置至少一个，通常不止一个 bean 由容器来管理。基于 XML 的配置元数据将这些 bean 配置为 `<bean/>` 元素，并放置到 `<bean/>` 元素内部。 Java 配置通常在 `@Configuration` 注解的类中使用 `@Bean` 注解到方法上。

这些 bean 定义对应于构成应用程序的实际对象。通常，您定义服务层对象，数据访问对象（DAO），展示对象，如 Struts Action 的情况下，基础设施的对象，如 Hibernate 的 SessionFactories，JMS Queues，等等。通常一个不配置细粒度域对象在容器中，因为它通常负责 DAO 和业务逻辑来创建和负载域对象。但是，你可以使用 Spring 和 AspectJ的集成配置在 IoC 容器的控制之外创建的对象。请参阅使用 AspectJ 在 Spring 中进行依赖关系注入域对象。

以下示例显示基于 XML 的配置元数据的基本结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

id 属性是一个字符串，唯一识别一个独立的 bean 定义。class 属性定义了 bean 的类型，并使用完整的类名。 id 属性的值是指协作对象。将XML 用于参照协作对象未在本例中示出;请参阅 依赖以获取更多信息。

实例化容器

实例化 Spring IoC 容器是直截了当的。提供给 ApplicationContext构造器的路径就是实际的资源字符串，使容器装入从各种外部资源的配置元数据，如本地文件系统， Java CLASSPATH，等等。

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

当你了解 Spring 的 IoC 容器，你可能想知道更多关于 Spring 的Resource 抽象，如第7章，资源，它提供了一种方便的从一个 URI 语法定义的位置读取一个InputStream 描述。特别地，资源路径被用作构建应用程序的上下文，详见第7.7节，“应用环境和资源的路径”。

下面的例子显示了服务层对象（services.xml中）配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- services -->

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStore
ServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for services go here -->

</beans>
```

下面的例子显示了数据访问对象 daos.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao
">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

在上面的例子中，服务层由类 PetStoreServiceImpl，以及类型的两个数据访问对象 JpaAccountDao 和 JpaItemDao（基于JPA对象/关系映射标准）组成。property name 元素是指 JavaBean 属性的名称，而 ref 元素引用另一个 bean 定义的名称。 id 和 ref 元素之间的这种联系表达了合作对象之间的依赖关系。对于配置对象的依赖关系的详细信息，请参阅依赖。

撰写基于XML的配置元数据

它可以让 bean 定义跨越多个 XML 文件,这样做非常有用。通常，每个单独的 XML 配置文件代表你的架构一个逻辑层或模块。

您可以使用应用程序上下文构造从所有这些 XML 片段加载 bean 定义。这个构造函数的多个 Resource 位置，作为上一节中被证明。另外，使用 `<import/>` 元素的一个或多个出现，从另一个或多个文件加载 bean 定义。 例如：

```
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

上面的例子中，使用了3个文件：services.xml，messageSource.xml 及 themeSource.xml来加载外部 bean 定义。所有位置路径是相对于导入文件的，因此 services.xml 是必须和导入文件在同一目录或类路径中的位置，而 messageSource.xml 和 themeSource.xml 来必须在导入文件的 resources 以下位置。正如你所看到的，前面的斜线被忽略，但考虑到这些路径是相对的，它更好的形式是不使用斜线。该文件的内容被导入，包括顶级 `<beans />` 元素，必须根据 Spring Schema 是有效的XML bean 定义。

这是可能的，但不推荐，引用在使用相对“../”的路径的父目录中的文件。这样将创建一个文件，该文件是当前应用程序之外的依赖。特别是，该引用不推荐*“classpath：” URL*（例如，*“classpath:../services.xml”*），在运行时解决过程中选择了“就近”的类路径的根，然后查找到它的父目录。类路径配置的变化可能导致不同的，不正确的目录的选择。 您可以随时使用完全合格的资源位置，而不是相对路径：例如，*file:C:/config/services.xml"* 或 *"classpath:/config/services.xml"*。但是，要知道，你这是是在耦合应用程序的配置到特定的绝对位置。通常优选间接的方式应对这种绝对路径，例如，通过*“${...}”*在运行时解决了对*JVM*系统属性占位符。

使用容器

ApplicationContext 是能够保持 bean 定义以及相互依赖关系的高级工厂接口。使用方法 T getBean(String name, Class requiredType)就可以取得 bean 的实例。

ApplicationContext 中可以读取 bean 定义并访问它们，如下所示：

```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);

// use configured instance
List<String> userList = service.getUsernameList();
```

您可以使用 getBean() 方法来获取 bean 的实例。 ApplicationContext 接口有一些其他的方法来获取 bean，但理想的应用程序代码不应该使用它们。事实上，你的应用程序代码不应该调用的getBean() 方法，因此在 Spring 的 API 没有依赖性的。例如，Spring如何与Web框架的集成提供了依赖注入各种Web框架类，如控制器和 JSF 管理的bean。

Bean 总览

Spring IoC 容易管理一个或者多个 bean。 bean 由应用到到容器的配置元数据创建,例如,在 XML 中定义 `<bean/>` 的形式。

容器内部,这些 bean 定义表示为 BeanDefinition 对象,其中包含(其他信息)以下元数据:

- 限定包类名称:典型的实际实现是定义 bean 的类。
- bean 行为配置元素,定义了容器中的Bean应该如何行为(范围、生命周期回调,等等)。
- bean 需要引用其他 bean 来完成工作,这些引用也称为合作者或依赖关系。
- 其他配置设置来设置新创建的对象,例如,连接使用 bean 的数量管理连接池,或者池的大小限制。

以下是每个 bean 定义的属性。

Table 6.1. The bean definition

属性	解释
class	Section 6.3.2, “Instantiating beans”
name	Section 6.3.1, “Naming beans”
scope	Section 6.5, “Bean scopes”
constructor arguments	Section 6.4.1, “Dependency Injection”
properties	Section 6.4.1, “Dependency Injection”
autowiring mode	Section 6.4.5, “Autowiring collaborators”
lazy-initialization mode	Section 6.4.4, “Lazy-initialized beans”
initialization method	the section called “Initialization callbacks”
destruction method	the section called “Destruction callbacks”

除了包含的信息里面 bean 定义的如何创建一个特定的bean, ApplicationContext 的实现还允许由用户注册现有创建在容器之外的现有对象。这是通过访问 ApplicationContext 的 BeanFactory 的 getBeanFactory() 方法 返回 BeanFactory 的实现 DefaultListableBeanFactory 。DefaultListableBeanFactory 支持这种通过 registerSingleton(..) 和registerBeanDefinition(..) 方法来注册。然而,典型的应用程序只能通过元数据定义的 bean 来定义。

需要尽早注册 Bean 元数据和手动使用单例的实例,这是为了使容器正确推断它们在自动装配和其他内省的步骤。虽然覆盖现有的元数据和现有的单例实例在某种程度上是支持的,新 bean 在运行时(同时动态访问工厂)注册不是官方支持,可能会导致并发访问 bean 容器中的异常和/或不一致的状态。

命名bean

每个 bean 都有一个或多个标识符。这些标识符在容器托管 bean 必须是唯一的。bean 通常只有一个标识符,但如果它需要不止一个,可以考虑额外的别名。

在基于 xml 的配置中,您可以使用 id 和(或)名称属性指定 bean 标识符。(id 属性允许您指定一个 id。通常这些名字使用字母数字(“myBean”、“fooService”,等等),但可以包含特殊字符。如果你想使用bean别名,您可以在 `name` 属性上定义它们,由逗号(,),分号(;),或白色空格进行分隔。作为一个历史因素的要注意,在 Spring 3.1 版本之前,id 属性被定义为 xsd:ID类型,它限制可能的字符。3.1,它被定义为一个 xsd:string 类型。注意,bean id 独特性仍由容器执行,虽然不再由 XML 解析器。

你不需要提供一个 bean 的名称或id。如果没有显式地提供名称或id, 容器生成一个唯一的名称给 bean 。然而,如果你想引用 bean 的名字,通过使用 ref 元素或使用 Service Locator（服务定位器）风格查找,你必须提供一个名称。不使用名称的原因是，内部 bean 和自动装配的合作者。

bean 名约定

约定是使用标准 *Java* 实例字段名称命名 *bean* 时的约定。也就是说,*bean* 名称开始以小写字母开头,后面采用“骆峰式”。例
如*“accountManager”*、*“accountService’,‘userDao’,‘loginController’*,等等。

一致的*beans*命名可以让您的配置容易阅读和理解，如果你正在使用*Spring AOP*，当你通过 *bean* 名称应用到 *advice* 时，这会对你帮助很大。

bean 的别名

在对 bean 定义时，除了使用 id 属性指定一个唯一的名称外，为了提供多个名称，需要通过 name 属性加以指定，所有这个名称都指向同一个bean，在某些情况下提供别名非常有用，比如为了让应用每一个组件都能更容易的对公共组件进行引用。然而，在定义 bean 时就指定所有的别名并不总是很恰当。有时我们期望能够在当前位置为那些在别处定义的bean引入别名。在XML配置文件中，可以通过 `<alias/>` 元素来完成 bean 别名的定义，例如：

```
<alias name="fromName" alias="toName"/>
```

上面示例中，在同一个容器中名为 fromName 的 bean 定义，在增加别名定义后，也可以用 toName 来引用。

例如，在子系统 A 中通过名字 subsystemA-dataSource 配置的数据源。在子系统B中可能通过名字 subsystemB-dataSource 来引用。当两个子系统构成主应用的时候，主应用可能通过名字 myApp-dataSource 引用数据源，将全部三个名字引用同一个对象，你可以将下面的别名定义添加到应用配置中：

```
<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>
<alias name="subsystemA-dataSource" alias="myApp-dataSource" />
```

现在每个子系统和主应用都可以通过唯一的名称来引用相同的数据源，并且可以保证他们的定义不与任何其他的定义冲突。

基于 *Java* 的配置

如果你想使用基于 *Java* 的配置，*@Bean* 注解可以用来提供别名，详细信息请看 *Section 6.12.3, “Using the @Bean annotation”*

实例化bean

bean 定义基本上就是用来创建一个或多个对象的配置，当需要一个 bean 的时候，容器查看配置并且根据 bean 定义封装的配置元数据创建（或获取）一个实际的对象。

如果你使用基于 XML 的配置，你可以在 `<bean/>` 元素通过 class 属性来指定对象的类型。这个 class 属性，实际上是 BeanDefinition 实例中的一个 Class 属性。这个 class 属性通常是必须的（例外情况，查看 “使用实例工厂方法实例化” 章节和 Section 6.7, “Bean定义的继承”），使用 Class 属性的两种方式：

- 通常情况下，直接通过反射调用构造方法来创建 bean，和在 Java 代码中使用 new 有点像。
- 通过静态工厂方法创建，类中包含静态方法。通过调用静态方法返回对象的类型可能和 Class 一样，也可能完全不一样。

内部类名。如果你想配置使用静态的内部类，你必须用内部类的二进制名称。例如，在 *com.example* 包下有个 *Foo* 类，这里类里面有个静态的内部类*Bar*，这种情况下*bean*定义的*class*属性应该*…com.example.Foo$Bar* 注意，使用*$*字符来分割外部类和内部类的名称。

通过构造函数实例化

当你使用构造方法来创建 bean 的时候，Spring 对类来说并没有什么特殊。也就是说，正在开发的类不需要实现任何特定的接口或者以特定的方式进行编码。但是，根据你使用那种类型的 IoC 来指定 bean，你可能需要一个默认（无参）的构造方法。

Spring IoC 容器可以管理几乎所有你想让它管理的类，它不限于管理POJO。大多数 Spring 用户更喜欢使用 POJO（一个默认无参的构造方法和setter,getter方法）。但在容器中使用非 bean 形式(non-bean style)的类也是可以的。比如遗留系统中的连接池，很显然它与 JavaBean规范不符，但 Spring 也能管理它。

当使用基于XML的元数据配置文件，可以这样来指定 bean 类：

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

给构造方法指定参数以及为bean实例化设置属性将在后面的依赖注入中详细说明。

使用静态工厂方法实例化

当采用静态工厂方法创建 bean 时，除了需要指定 class 属性外，还需要通过 factory-method 属性来指定创建 bean 实例的工厂方法。Spring将调用此方法(其可选参数接下来介绍)返回实例对象，就此而言，跟通过普通构造器创建类实例没什么两样。

下面的 bean 定义展示了如何通过工厂方法来创建bean实例。注意，此定义并未指定返回对象的类型，仅指定该类包含的工厂方法。在此例中，createInstance() 必须是一个 static 方法。

```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

```
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

给工厂方法指定参数以及为bean实例设置属性的详细内容请查阅依赖和配置详解。

使用实例工厂方法实例化

与通过 静态工厂方法 实例化类似，通过调用工厂实例的非静态方法进行实例化。 使用这种方式时，class属性置为空，而factory-bean属性必须指定为当前(或其祖先)容器中包含工厂方法的bean的名称，而该工厂bean的工厂方法本身必须通过factory-method属性来设定。

```
<!-- 工厂bean，包含createInstance()方法 -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 其他需要注入的依赖项 -->
</bean>

<!-- 通过工厂bean创建的ben -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();
    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

一个工厂类也可以有多个工厂方法，如下代码所示：

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 其他需要注入的依赖项 -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();
    private static AccountService accountService = new AccountServiceImpl();

    private DefaultServiceLocator() {}

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }

}
```

这种做法表明工厂bean本身也可以通过依赖注入（DI）进行管理配置。查看依赖和配置详解。

在*Spring*文档中，*factory bean*是指在*Spring*容器中配置的工厂类通过 实例 或 静态 工厂方法来创建对象。相比而言, *FactoryBean (*注意大小写*)* 代表了*Spring*中特定的 *FactoryBean*

Dependencies

一般情况下企业应用不会只有一个对象（或者是Spring Bean）。甚至最简单的应用都要多个对象来协同工作来让终端用户看到一个完整的应用的。下一部分将解释开发者如何从仅仅定义单独的Bean，到让这些Bean在一个应用中协同工作。

Dependency Injection

依赖注入是一个让对象只通过构造参数，工厂方法的参数或者配置的属性来定义他们的依赖的过程。这些依赖也是对象所需要协同工作的对象。容器会在创建Bean的时候注入这些依赖。整个过程完全反转了由Bean自己控制实例化或者引用依赖，所以这个过程也称之为控制反转。

当使用了依赖注入的准则以后，会更易于管理和解耦对象之间的依赖，使得代码更加的简单。对象不再关注依赖，也不需要知道依赖类的位置。这样的话，开发者的类更加易于测试，尤其是当开发者的依赖是接口或者抽象类的情况，开发者可以轻易在单元测试中mock对象。

依赖注入主要使用两种方式，一种是基于构造函数的注入，另一种的基于Setter方法的依赖注入。

Constructor-based dependency Injection

基于构造函数的依赖注入是由IoC容器来调用类的构造函数，构造函数的参数代表这个Bean所依赖的对象。跟调用带参数的静态工厂方法基本一样。下面的例子展示了一个类通过构造函数来实现依赖注入的。需要注意的是，这个类没有任何特殊的地方，只是一个简单的，不依赖于任何容器特殊接口，基类或者注解的普通类。

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on a MovieFinder
    private MovieFinder movieFinder;

    // a constructor so that the Spring container can inject a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...

}
```

构造函数的参数解析

构造函数的参数解析是通过参数的类型来匹配的。如果在Bean的构造函数参数不存在歧义，那么构造器参数的顺序也就是就是这些参数实例化以及装载的顺序。参考如下代码：

```
package x.y;

public class Foo {

    public Foo(Bar bar, Baz baz) {
        // ...
    }

}
```

假设 `Bar` 和 `Baz` 在继承层次上不相关，也没有什么歧义的话，下面的配置完全可以工作正常，开发者不需要再去 `<constructor-arg>` 元素中指定构造函数参数的索引或类型信息。

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>

    <bean id="bar" class="x.y.Bar"/>

    <bean id="baz" class="x.y.Baz"/>
</beans>
```

当引用另一个Bean的时候，如果类型确定的话，匹配会工作正常（如上面的例子）.当使用简单的类型的时候，比如说 `<value>true</value>` ，Spring IoC容器是无法判断值的类型的，所以是无法匹配的。考虑代码如下：

```
package examples;

public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }

}
```

在上面代码这种情况下，容器可以通过使用构造函数参数的 `type` 属性来实现简单类型的匹配。比如：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

或者使用 `index` 属性来指定构造参数的位置，比如：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

这个索引也同时是为了解决构造函数中有多个相同类型的参数无法精确匹配的问题。需要注意的是，索引是基于0开始的。 开发者也可以通过参数的名称来去除二义性。

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

需要注意的是,做这项工作的代码必须启用了调试标记编译,这样Spring才可以从构造函数查找参数名称。开发者也可以使用 `@ConstructorProperties` 注解来显式声明构造函数的名称，比如如下代码：

```
package examples;

public class ExampleBean {

    // Fields omitted

    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }

}
```

Setter-based dependency injection

基于Setter函数的依赖注入则是容器会调用Bean的无参构造函数，或者无参数的工厂方法，然后再来调用Setter方法来实现的依赖注入。

下面的例子展示了使用Setter方法进行的依赖注入，下面的类对象只是简单的POJO对象，不依赖于任何Spring的特殊的接口，基类或者注解。

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can inject a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...

}
```

`ApplicationContext` 所管理Bean对于基于构造函数的依赖注入，或者基于Setter方式的依赖注入都是支持的。同时也支持使用Setter方式在通过构造函数注入依赖之后再次注入依赖。开发者在 `BeanDefinition` 中可以使用 `PropertyEditor` 实例来自由选择注入的方式。然而，大多数的开发者并不直接使用这些类，而是跟喜欢XML形式的 `bean` 定义，或者基于注解的组件（比如使用 `@Component` ， `@Controller` 等）或者在配置了 `@Configuration` 的类上面使用 `@Bean` 的方法。

> 基于构造函数还是基于Setter方法？ 因为开发者可以混用两者，所以通常比较好的方式是通过构造函数注入必要的依赖通过Setter方式来注入一些可选的依赖。其中，在Setter方法上面的 `@Required` 注解可用来构造必要的依赖。 Spring队伍推荐基于构造函数的注入，因为这种方式会促使开发者将组件开发成不可变对象而且确保了注入的依赖不为 `null` 。而且，基于构造函数的注入的组件被客户端调用的时候也是完全构造好的。当然，从另一方面来说，过多的构造函数参数也是非常差的代码方式，这种方式说明类貌似有了太多的功能，最好重构将不同职能分离。 基于Setter的注入只是用于可选的依赖，但是也最好配置一些合理的默认值。否则，需要对代码的依赖进行非NULL的检查了。基于Setter方法的注入有一个便利之处在于这种方式的注入是可以进行重配置和重新注入的。 依赖注入的两种风格适合大多数的情况，但是有时使用第三方的库的时候，开发者可能并没有源码，而第三方的代码也没有setter方法，那么就只能使用基于构造函数的依赖注入了。

Dependency resolution process

容器对Bean的解析如下：

- 创建并根据描述的元数据来实例化 `ApplicationContext` 。配置元数据可以通过XML，Java 代码，或者注解。
- 每一个Bean的依赖通过构造函数参数或者属性或者静态工厂方法的参数等来表示。这些依赖会在Bean创建的的时候注入和装载。
- 每一个属性或者构造函数的参数都是实际定义的值或者引用容器中其他的Bean。
- 每一个属性或者构造参数可以根据其指定的类型转换而成。Spring也可以将String转成默认的Java内在的类型，比如 `int` , `long` , `String` , `boolean` 等。

Spring容器会在容器创建的时候针对每一个Bean进行校验。然而，Bean的属性在Bean没有真正创建的时候是不会配置进去的。单例类型的Bean是容器创建的时候配置成预实例状态的。Bean的 `Scope` 在后续有介绍。其他的Bean都只有在请求的时候，才会创建。显然创建Bean对象会有一个依赖的图。这个图表示Bean之间的依赖关系，容器根据此来决定创建和配置Bean的顺序。

> 循环依赖 如果开发者主要使用基于构造函数的依赖注入，那么很有可能出现一个循环依赖的场景。 比如说：类A在构造函数中依赖于类B的实例，而类B的构造函数依赖类A的实例。如果你这么配置类A和类B相互注入的话，Spring IoC容器会发现这个运行时的循环依赖，并且抛出 `BeanCurrentlyInCreationException` 。 开发者可以通过使用Setter方法来配置依赖注入，这样可以解决这个问题。或者就不使用基于构造函数的依赖注入，仅仅使用基于Setter方法的依赖注入。换言之，尽管不推荐，但是开发者可以将循环依赖配置为基于Setter方法的依赖注入。

开发者可以相信Spring能正确处理Bean。Spring能够在加载的过程中发现配置的问题，比如引用到不存在的Bean或者是循环依赖。Spring会尽可能晚的在Bean创建的时候装载属性或者解析依赖。这也意味着Spring容器加载正确后会在Bean注入依赖出错的时候抛出异常。比

如，Bean抛出缺少属性或者属性不合法。这延迟的解析也是为什么 `ApplicationContext` 的实现会令单例Bean处于预实例化状态。这样，通过 `ApplicationContext` 的创建，可以在真正使用Bean之前消耗一些内存代价发现配置的问题。开发者也可以覆盖默认的行为让单例Bean延迟加载，而不是处于预实例化状态。 如果不存在循环依赖的话，Bean所引用的依赖会优先完全构造依赖的。举例来说，如果Bean A依赖于Bean B，那么Spring IoC容器会先配置Bean B，然后调用Bean A的Setter方法来构造Bean A。换言之，Bean先会实例化，然后注入依赖，然后才是相关的生命周期方法的调用。

Examples of dependency injection

下面的例子会使用基于XML配置的元数据，然后使用Setter方式进行依赖注入。代码如下：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>

    <!-- setter injection using the neater ref attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }

}
```

在上面的例子当中，Setter方法的声明和XML文件中相一致，下面的例子是基于构造函数的依赖注入

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

    <!-- constructor injection using the neater ref attribute -->
    <constructor-arg ref="yetAnotherBean"/>

    <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public ExampleBean(
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }

}
```

在Bean定义之中的构造函数参数就是用来构造 `ExampleBean` 的依赖。

下面的例子，是通过静态的工厂方法来返回Bean实例的。

```
<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">
    <constructor-arg ref="anotherExampleBean"/>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    // a private constructor
    private ExampleBean(...) {
        ...
    }

    // a static factory method; the arguments to this method can be
    // considered the dependencies of the bean that is returned,
    // regardless of how those arguments are actually used.
    public static ExampleBean createInstance (
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

        ExampleBean eb = new ExampleBean (...);
        // some other operations...
        return eb;
    }

}
```

工厂方法的参数，也是通过 `<constructor-arg/>` 标签来指定的，和基于构造函数的依赖注入是一致的。之前有提到过，返回的类型不需要跟 `exampleBean` 中的 `class` 属性一致的， `class` 指定的是包含工厂方法的类。当然了，上面的例子是一致的。使用 `factory-bean` 的实例工厂方法构造Bean的，这里就不多描述了。

7.4.2 Dependencies and configuration in detail

如前文所述，开发者可以通过定义Bean的依赖的来引用其他的Bean或者是一些值的，Spring基于XML的配置元数据通过支持一些子元素 `<property/>` 以及 `<constructor-arg/>` 来达到这一目的。

Straight values (primitives, Strings, and so on)

元素 `<property/>` 有 `value` 属性来对人友好易读的形式配置一个属性或者构造参数。Spring的便利之处就是用来将这些字符串的值转换成指定的类型。

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method
="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

下面的例子使用的p命名空间，是更为简单的XML配置。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/mydb"
        p:username="root"
        p:password="masterkaoli"/>

</beans>
```

上面的XML更为的简洁，但是因为属性的类型是在运行时确定的，而非设计时确定的，所以除非使用IntelliJ IDEA或者Spring Tool Suite这些工具才能在定义Bean的时候自动完成属性配置。当然很推荐使用这些IDE。

开发者也可以定义一个 `java.util.Properties` 实例，比如：

```
<bean id="mappings"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

    <!-- typed as a java.util.Properties -->
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver
            jdbc.url=jdbc:mysql://localhost:3306/mydb
        </value>
    </property>
</bean>
```

Spring的容器会将 `<value/>` 里面的文本通过使用JavaBean的 `PropertyEditor` 机制转换成一个 `java.util.Properties` 实例。这也是一个捷径，也是一些Spring团队更喜欢使用嵌套的 `<value/>` 元素而不是 `value` 属性风格。

idref元素 `idref` 元素是一种简单的提前校验错误的方式，通过id来关联容器中的其他的Bean的方式。

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

上述的Bean的定义在运行时，和如下定义是完全一致的。

```
<bean id="theTargetBean" class="..." />

<bean id="client" class="...">
    <property name="targetName" value="theTargetBean" />
</bean>
```

第一种方式是更值得提倡的，因为使用了 `idref` 标签，会是的容器在部署阶段就针对Bean进行校验，确保Bean一定存在。而第二个版本的话，是没有任何校验的。只有实际上引用了Bean `client` ，在实例化client的时候才会发现。如果 `client` 是一个 `prototype` 的Bean，那么类似拼写之类的错误会在容器部署以后很久才能发现。

> `idref` 元素的 `local` 属性在4.0以后的xsd中已经不再支持了，而是使用了 `bean` 引用。如果更新了版本的话，需要将 `idref local` 引用都转换成 `idref bean` 即可。

References to other beans (collaborators)

`ref` 元素在 `<constructor-arg/>` 或者 `<property/>` 中的一个终极标签。开发者可以通过这个标签配置一个Bean来引用另一个Bean。当需要引用一个Bean的时候，被引用的Bean会先实例化，然后配置属性，也就是引用的依赖。如果该Bean是单例Bean的话，那么该Bean会早由容器初始化。最终的引用另一个对象的所有引用。Bean的范围以及校验取决于开发者是否通过 `bean` , `local` , `parent` 这些属性来指定对象的id或者name属性。

通过指定Bean的 `bean` 属性中的 `<ref/>` 来指定依赖是最常见的一种方式，可以引用容器或者父容器中的Bean，无论是否在同一个XML文件定义都可以引用。其中 `bean` 属性中的值可以和其他引用Bean中的 `id` 属性一致，或者和其中的一个 `name` 属性一致的。

```
<ref bean="someBean"/>
```

通过指定Bean的 `parent` 属性会创建一个引用到当前容器的父容器之中。 `parent` 属性的值可以跟跟目标Bean的 `id` 属性一致，或者和目标Bean的 `name` 属性中的一个一致，目标Bean必须是当前引用目标Bean容器的父容器。开发者一般只有在存在层次化容器，并且希望通过代理来包裹父容器中一个存在的Bean的时候才会用到这个属性。

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService" <!-- bean name is the same as the parent bean -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/> <!-- notice how we refer to the parent bean -->
    </property>
    <!-- insert other configuration and dependencies as required here -->
</bean>
```

> 与 `idref` 标签一样， `ref` 元素中的 `local` 标签在xsd 4.0以后已经不再支持了，开发者可以通过将已存在的 `ref local` 改为 `ref bean` 来完成更新Spring。

Inner beans

定义在 `<bean/>` 元素的 `<property/>` 或者 `<constructor-arg/>` 元素之内的Bean叫做内部*Bean*

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target bean
inline -->
    <property name="target">
        <bean class="com.example.Person"> <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

内部Bean的定义是不需要指定id或者名字的。如果指定了，容器也不会用之作为分别Bean的区分标识。容器同时也会无视内部Bean的 `scope` 标签：内部Bean 总是 匿名的，而且 总是 随着外部的Bean同时创建的。开发者是无法将内部的Bean注入到外部Bean以外的其他Bean的。

当然，也有可能从指定范围接收到破坏性回调。比如：一个请求范围的内部Bean包含了一个单例的Bean，那么内部Bean实例会绑定到包含的Bean，而包含的Bean允许访问到 `request` 的 `scope` 生命周期。这种场景不常见，内部Bean通常只是共享它的外部Bean。

Collections

在 `<list/>` , `<set/>` , `<map/>` 和 `<props/>` 元素中，开发者可以配置Java集合类型 `List` , `Set` , `Map` 以及 `Properties` 的属性和参数。

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

当然，map的key或者value，或者是集合的value都可以配置为下列之中的一些元素：

```
bean | ref | idref | list | set | map | props | value | null
```

集合合并 Spring的容器也支持来合并集合。开发者可以定义一个父样式的 `<list/>` , `<map/>` , `<set/>` 或者 `<props/>` ，同时有子样式的 `<list/>` , `<map/>` , `<set/>` 或者 `<props/>` 继承并且覆盖父集合。也就是说，子集合的值是父元素和子元素集合的合并值。比如下面的例子。

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
            <props>
                <prop key="administrator">administrator@example.com</prop>
                <prop key="support">support@example.com</prop>
            </props>
        </property>
    </bean>
    <bean id="child" parent="parent">
        <property name="adminEmails">
            <!-- the merge is specified on the child collection definition -->
            <props merge="true">
                <prop key="sales">sales@example.com</prop>
                <prop key="support">support@example.co.uk</prop>
            </props>
        </property>
    </bean>
<beans>
```

可以发现，我们在 `child` Bean上使用了 `merge=true` 属性。当 `child` Bean由容器初始化且实例化的时候，其实例中包含的 `adminEmails` 集合就是 `child` 的 `adminEmails` 以及 `parent` 的 `adminEmails` 集合。如下：

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

`child` 的 `Properties` 集合的值继承了 `parent` 的 `<props/>` ， `child` 的值也支持重写 `parent` 的值。

这个合并的行为和 `<list/>` , `<map/>` 以及 `<set/>` 之类的集合类型的行为是类似的。 `<list/>` 的特定的例子中，与 `List` 集合类型类似，有隐含的 `ordered` 概念的。所有的父元素里面的值，是在所有孩子元素的值之前的。但是像 `Map` , `Set` 或者 `Properties` 的集合类型，是不存在顺序的。

集合合并的限制

开发者是不能够合并不同类型的集合的（比如 `Map` 和 `List` 合并），如果开发者这么做，会抛出异常。 `merge` 的属性是必须特指到更低级或者继承者，子节点的定义上。特指 `merge` 属性到父集合的定义上是冗余的，而且在合并上也没有任何效果。

强类型集合 在Java 5以后，开发者可以使用强类型的集合了。也就是，开发者可以声明一个 `Collection` 类型，然后这个集合只包含 `String` 元素（举例来说）。如果开发者通过Spring来注入强类型的 `Collection` 到Bean中，开发者就可以利用Spring的类型转换支持来做到。

```
public class Foo {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
```

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

当 `foo` 的属性 `accounts` 准备注入的时候， `accounts` 的泛型信息 `Map<String, Float>` 就会通过反射拿到。这样，Spring 的类型转换系统能够识别不同的类型，如上面的例子 `Float` 然后将字符串的值 `9.99, 2.75` 以及 `3.99` 转换成对应的 `Float` 类型。

Null and empty string values

Spring将会将属性的空参数，直接当成空字符串来处理。下面的基于XML的元数据配置就会将email属性配置为 `String` 的值为 `""`

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

上面的例子和下列JAVA代码是一致的。

```
exampleBean.setEmail("")
```

而 `<null/>` 元素来处理 `null` 的值。如下：

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

上面的代码和下面的Java代码是一样的效果：

```
exampleBean.setEmail(null)
```

XML shortcut with the p-namespace

p命名空间令开发者可以使用`bean`的属性，而不用使用嵌套的`<property/>`元素，就能描述开发者想要注入的依赖。

Spring是支持基于XML的格式化的命名空间扩展的。本节讨论的`beans`的配置都是基于XML的，p命名空间并不是定义在XSD文件，而是定义在Spring Core之中的。

下面展示了两种XML片段是同样的解析结果：第一个使用的标准的XML格式，而第二种使用了p命名空间。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="foo@bar.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="foo@bar.com"/>
</beans>
```

上面的例子在Bean中展示了email属性的定义。这种定义告知Spring这是一个属性的声明。如前面所描述，p命名空间并没有标准模式定义，所以你可以配置属性的名字为依赖名字。

下面的例子包括了2个Bean定义，引用了另外的Bean：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

从上述的例子中可以看出， `john-modern` 不止包含一个属性，也同时使用了特殊的格式来声明一个引用指向另一个Bean。第一个Bean定义使用的是 `<property name="spouse" ref="jane"/>` 来创建的Bean引用到另外一个Bean，而第二个Bean的定义使用了 `p:spouse-ref="jane"` 来作为一个指向Bean的引用。在这个例子中 `spouse` 是属性的名字，而 `-ref` 部分表名这个依赖不是直接的类型，而是引用另一个Bean。

> p命名空间并不同标准的XML格式一样灵活。比如，声明属性的引用可能和一些以 `Ref` 结尾的属性相冲突，而标准的XML格式就不会。Spring团队推荐开发者能够和团队商量一下，要使用哪一种方式，而不要同时使用3种方法。

XML shortcut with the c-namespace

与p命名空间类似，c命名空间是在Spring 3.1首次引入的，c命名空间允许内联的属性来配置构造参数而不用使用 `constructor-arg` 元素。 下面就是一个使用了c命名空间的例子：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>

    <!-- traditional declaration -->
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
        <constructor-arg value="foo@bar.com"/>
    </bean>

    <!-- c-namespace declaration -->
    <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.co
m"/>

</beans>
```

`c:` 命名空间使用了和 `p:` 命名空间相类似的方式（使用了 `-ref` 来配置引用）。而且，同样的，c命名空间也不是定义在XSD的模式之中（但是在Spring Core之中）。

在少数的例子之中，构造函数的参数名字并不可用（通常，如果字节码没有debug信息的编译），开发者可以使用下面的例子：

```
<!-- c-namespace index declaration -->
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz"/>
```

> 根据XML语法，索引的概念的存在要求使用 `_` 作为XML属性名字不能以数字开始。

实际上，构造函数的解析机制在匹配参数是很高效的，除非必要，Spring团队推荐在配置中使用命名空间。

Compound property names

开发者可以在配置属性的时候配置混合的属性，只要所有的组件路径（除了最后一个属性名字）不能为 `null` 。参考如下的定义。

```
<bean id="foo" class="foo.Bar">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

`foo` 有一个 `fred` 的属性，而其中 `fred` 属性有一个 `bob` 属性，而 `bob` 属性之中有一个 `sammy` 属性，那么最后这个 `sammy` 属性会配置为 `123` 。想要上述的配置能够生效， `fred` 属性需要有一个 `bob` 属性且在 `fred` 构造只是构造之后不为 `null` ，否则会抛出 `NullPointerException` 。

7.4.3 Using depends-on

如果一个Bean是另一个Bean的依赖的话，通常来说这个Bean也就是另一个Bean的属性之一。多数情况下，开发者可以在配置XML元数据的时候使用 `<ref/>` 标签。然而，有时Bean之间的依赖关系不是直接关联的。比如：需要调用类的静态实例化器来触发，类似数据库驱动注册。 `depends-on` 属性会使明确的强迫依赖的Bean在引用之前就会初始化。下面的例子使用 `depends-on` 属性来让表示单例Bean上的依赖的。

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

如果想要依赖多个Bean，可以提供多个名字作为 `depends-on` 的值，以逗号，空格，或者分号分割，如下：

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
    <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

> Bean中的 `depends-on` 属性可以同时指定一个初始化时间的依赖以及一个相应的销毁时依赖（单例Bean情况）。独立的定义了 `depends-on` 属性的Bean会优先销毁，优于被 `depends-on` 的Bean来销毁，这样 `depends-on` 可以控制销毁的顺序。

7.4.4 Lazy-initialized beans

默认情况下， `ApplicationContext` 会在实例化的过程中创建和配置所有的单例Bean。总的来说，这个预初始化是很不错的。因为这样能及时发现环境上的一些配置错误，而不是系统运行了很久之后才发现。如果这个行为不是迫切需要的，开发者可以通过将Bean标记为延迟加载就能阻止这个预初始化。延迟初始化的Bean会通知IoC不要让Bean预初始化而是在被引用的时候才会实例化。 在XML中，可以通过 `<bean/>` 元素的 `lazy-init` 属性来控制这个行为。如下：

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

当将Bean配置为上面的XML的时候，`ApplicationContext`之中的`lazy` Bean是不会随着`ApplicationContext`的启动而进入到预初始化状态的，而那些非延迟加载的Bean是处于预初始化的状态的。

然而，如果一个延迟加载的类是作为一个单例非延迟加载的Bean的依赖而存在的话，`ApplicationContext`仍然会在`ApplicationContext`启动的时候加载，因为作为单例Bean的依赖，会随着单例Bean的实例化而实例化。 开发者可以通过使用`<beans/>`的`default-lazy-init`属性来在容器层次控制Bean是否延迟初始化，比如：

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

7.4.5 Autowiring collaborators

Spring容器可以根据Bean之间的依赖关系自动装配。开发者可以令Spring通过`ApplicationContext`来来自动解析这些关联。自动的装载有很多的优点：

- 自动装载能够明显的减少指定的属性或者是构造参数。
- 自动装载可以扩展开发者的对象。比如说，如果开发者需要加一个依赖，依赖就能够不需要开发者特别关心更改配置就能够自动满足。这样，自动装载在开发过程中是极度高效的，不用明确的选择装载的依赖会使系统更加的稳定。

当使用基于XML的元数据配置的时候，开发者可以指定自动装配的方式。通过配置`<bean/>`元素的`autowire`属性就可以了。自动装载有如下四种方式，开发者可以指定每个Bean的装载方式，这样Bean就知道如何加载自己的依赖。

模式	解释
no	（默认）不装载。Bean的引用必须通过`ref`元素来指定。对于比较大项目的部署，不建议修改默认的配置，因为特指会加剧控制。在某种程度上来说，默认的形式也说明了系统的结构。
byName	通过名字来装配。Spring会查找所有的Bean直到名字和属性相同的一个Bean来进行装载。比如说，如果Bean配置为根据名字来自动装配，它包含了一个属性名字为`master` (也就是包含一个`setMaster(..)`方法)，Spring就会查找名字为`master`的Bean，然后用之装载
byType	如果需要自动装配的属性的类型在容器之中存在的话，就会自动装配。如果容器之中存在不止一个类型匹配的话，就会抛出一个重大的异常，说明开发者最好不要使用byType来自动装配那个Bean。如果没有匹配的Bean存在的话，不会抛出异常，只是属性不会配置。
构造函数	类似于*byType*的注入，但是应用的构造函数的参数。如果没有一个Bean的类型和构造函数参数的类型一致，那么仍然会抛出一个重大的异常

通过 *byType* 或者 构造函数 的自动装配方式，开发者可以装载数组和强类型集合。在如此的例子之中，所有容器之中的匹配指定类型的Bean会自动装配到Bean上来完成依赖注入。开发者可以自动装配key为 `String` 的强类型的 `Map` 。自动装配的 `Map` 值会包含所有的Bean实例值来匹配指定的类型， `Map` 的key会包含关联的Bean的名字。

Limitations and disadvantages of autowiring

自动装载如果在整个的项目的开发过程中使用，会工作的很好。但是如果不是全局使用，而只是用之来自动装配几个Bean的话，会很容易迷惑开发者。

下面是一些自动装配的劣势和限制

- 精确的 `property` 以及 `constructor-arg` 参数配置，会覆盖掉自动装配的配置。开发不能够自动装配所谓的简单属性，比如 `Primitive` 类型或者字符串。
- 自动装配并有精确装配准确。尽管如上面的表所描述，Spring会尽量小心来避免不必要的错误装配，但是Spring管理的对象关系仍然不如文档描述的那么精确。
- 装配的信息对开发者可见性不好，因为这一切都由Spring容器管理。
- 容器中的可能会存在很多的Bean匹配Setter方法或者构造参数。比如说数组，集合或者Map等。然而依赖却希望仅仅一个匹配的值，含糊的信息是无法解析的。如果没有独一无二的Bean，那么就会抛出异常。

在后面的场景，开发者有如下的选择

- 放弃自动装配有利于精确装配
- 可以通过配置 `autowire-candidate` 属性为 `false` 来阻止自动装配
- 通过配置 `<bean/>` 元素的 `primary` 属性为 `true` 来指定一个bean为主要的候选Bean
- 实现更多的基于注解的细粒度的装配配置。

Excluding a bean from autowiring

在每个Bean的基础之上，开发者可以阻止Bean来自动装配。在基于XML的配置中，可以配置 `<bean/>` 元素的 `autowire-candidate` 属性为 `false` 来做到这一点。容器在读取到这个配置后，会让这个Bean对于自动装配的结构中不可见（包括注解形式的配置比如 `@Autowired` ）

开发者可以通过模式匹配而不是Bean的名字来限制自动装配的候选者。最上层的 `<beans/>` 元素会在 `default-autowire-candidates` 属性中来配置多种模式。比如，限制自动装配候选者的名字以*Repository*结尾，可以配置 `*Repository` 。如果需要配置多种模式，只需要用逗号分隔开即可。当然Bean中如果配置了 `autowire-candidate` 的话，这个信息拥有更高的优先级。

上面的这些技术在配置那些不需要自动装配的Bean是很有效的。当然这并不是说这类Bean本身无法自动装配其他的Bean，而是说这些Bean不在作为自动装配依赖的候选了。

7.4.6 Method injection

在大多数的应用场景下，大多数的Bean都是单例的。当这个单例的Bean需要和另一个单例的或者非单例的Bean联合使用的时候，开发者只需要配置依赖的Bean为这个Bean的属性即可。但是有时会因为不同的Bean生命周期的不同而产生问题。假设单例的Bean A在每个方法调用中使用了非单例的Bean B。容器只会创建Bean A一次，而只有一个机会来配置属性。那么容器就无法给Bean A每次都提供一个新的Bean B的实例。

一个解决方案就是放弃一些IoC。开发者可以通过实现 `ApplicationContextAware` 接口令Bean A可以看到 `ApplicationContext` ，从而通过调用 `getBean("B")` 来在Bean A 需要新的实例的时候来获取到新的B实例。参考下面的例子：

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Object process(Map commandState) {
        // grab a new instance of the appropriate Command
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    protected Command createCommand() {
        // notice the Spring API dependency!
        return this.applicationContext.getBean("command", Command.class);
    }

    public void setApplicationContext(
            ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

上面的代码并不是让人十分满意，因为业务的代码已经与Spring框架耦合在了一起。Spring提供了一个稍微高级的点特性方法注入的方式，可以用来处理这种问题。

Lookup method injection

查找方法注入就是容器一种覆盖容器管理Bean的方法，来返回查找的另一个容器中的Bean的能力。查找方法通常就包含前面场景提到的Bean。Spring框架通过使用CGLIB库生成的字节码来动态生成子类来覆盖父类的方法实现方法注入。

> - 为了让这个动态的子类方案正常，那么Spring容器所需要继承的这个Bean不能是`final`的，而覆盖的方法也不能是`final`的。
> - 针对这个类的单元测试因为存在抽象方法，所以必须实现子类来测试
> - 组件扫描的所需的具体方法也需要具体类。
> - 一个关键的限制在于查找方法与工厂方法是不能协同工作的，尤其是不能和配置类之中的`@Bean`的方法，因为容器不在负责创建实例，而是创建一个运行时的子类。
> - 最后，被注入的到方法的对象不能被序列化。

看到前面的代码片段中的`CommandManager`类，我们发现发现Spring容器会动态的覆盖`createCommand()`方法。`CommandManager`类不在拥有任何的Spring依赖，如下：

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

在包含需要注入的方法的客户端类当中，注入的方法需要有如下的函数签名

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

如果方法为抽象，那么动态生成的子类会实现这个方法。否则，动态生成的子类会覆盖类中的定义的原方法。例如：

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
    <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="command"/>
</bean>
```

上面的commandManager在当它需要一个command Bean实例的时候就会调用自己的方法`createCommand()`。开发者一定要谨慎配置`command` Bean的为prototype类型的Bean。如果所需的Bean为单例的，那么这个方法注入返回的将都是同一个实例。

Arbitrary method replacement

从前面的描述中，我们知道查找方法是有能力来覆盖任何由容器管理的Bean的方法的。开发者最好跳过这一部分，除非一定需要使用这个功能。

通过配置基于XML的配置元数据，开发者可以使用`replaced-method`元素来替换一个存在的方法的实现。考虑如下情况：

```
public class MyValueCalculator {

    public String computeValue(String input) {
        // some real code...
    }

    // some other methods...

}
```

一个实现了`org.springframework.beans.factory.support.MethodReplacer`接口的类会提供一个新方法的定义。

```
/**
 * meant to be used to override the existing computeValue(String)
 * implementation in MyValueCalculator
 */
public class ReplacementComputeValue implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

如果需要覆盖Bean的方法需要配置XML如下：

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
    <!-- arbitrary method replacement -->
    <replaced-method name="computeValue" replacer="replacementComputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

开发者可以使用更多的`<replaced-method>`中的`<arg=type/>`元素来指定需要覆盖的方法。当需要覆盖的方法存在重载方法时，指定参数才是必须的。为了方便起见，字符串的类型是会匹配如下类型，完全等同于`java.lang.String`。

```
java.lang.String
String
Str
```

因为通常来说参数的个数已经足够区别不同的方法了，这种快捷的写法可以省去很多的代码。

Bean scopes

当开发者定义Bean的时候，同时也会定义了该如何创建Bean实例。这些具体创建的过程是很重要的，因为只有通过对这些过程的配置，开发者才能创建实例对象。

开发者不仅可以控制注入不同的依赖到Bean之中，也可以配置Bean的作用域。这种方法是非常强大而且弹性也非常好的。开发者可以通过配置来指定对象的作用域，而不用在Java类层次上来配置。Bean可以配置多种作用域。 Spring框架支持5种作用域，有三种作用域是当开发者使用基于web的 `ApplicationContext` 的时候才生效的。

下面就是Spring直接支持的作用域了，当然开发者也可以自己定制作用域。

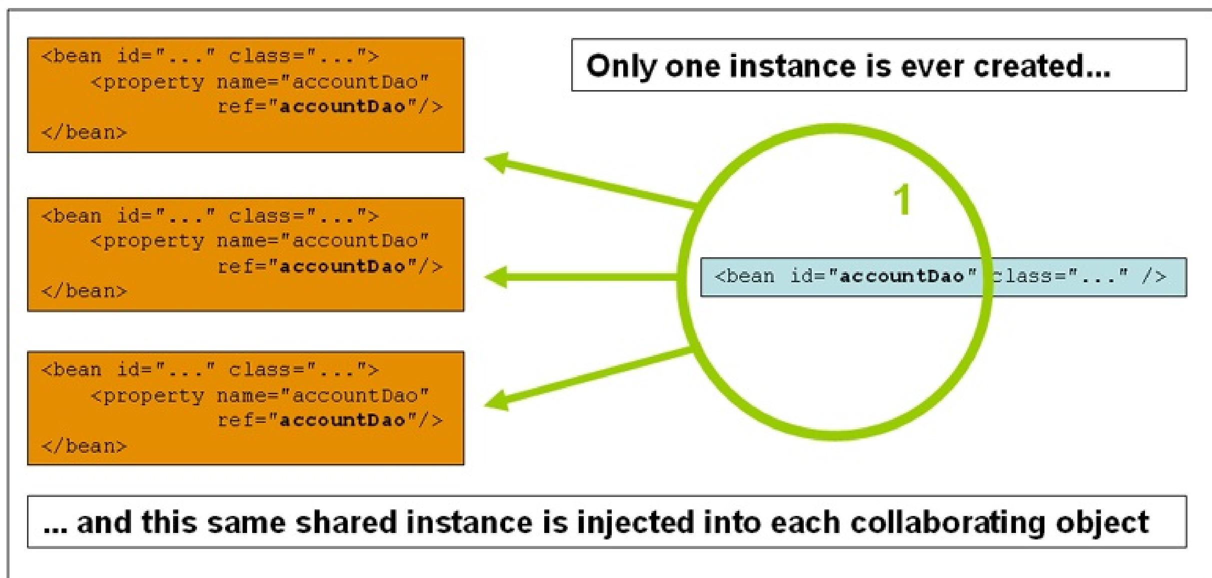
作用域	描述
单例 (singleton)	（默认）每一个Spring IoC容器都拥有唯一的一个实例对象
原型（prototype）	一个Bean定义可以创建任意多个实例对象
请求（request）	一个HTTP请求会产生一个Bean对象，也就是说，每一个HTTP请求都有自己的Bean实例。只在基于web的Spring `ApplicationContext` 中可用
会话（session）	限定一个Bean的作用域为HTTP `session` 的生命周期。同样，只有基于web的Spring `ApplicationContext` 才能使用
全局会话（global session）	限定一个Bean的作用域为全局HTTP `Session` 的生命周期。通常用于门户网站场景，同样，只有基于web的Spring `ApplicationContext` 可用
应用 (application)	限定一个Bean的作用域为 `ServletContext` 的生命周期。同样，只有基于web的Spring `ApplicationContext` 可用

> 在Spring 3.0中，线程作用域是可用的，但不是默认注册的。想了解更多的信息，可以参考本文后面关于 `SimpleThreadScope` 的文档。想要了解如何注册这个或者其他的自定义的作用域，可以参考后面的内容。

The singleton scope

单例Bean全局只有一个共享的实例，所有将单例Bean作为依赖的情况下，容器返回将是同一个实例。

换言之，当开发者定义一个Bean的作用域为单例时，Spring IoC容器只会根据Bean定义来创建该Bean的唯一实例。这些唯一的实例会缓存到容器中，后续针对单例Bean的请求和引用，都会从这个缓存中拿到这个唯一的实例。



Spring的单例Bean和与设计模式之中的所定义的单例模式是有所区别的。设计模式中的单例模式是将一个对象的作用域硬编码的，一个ClassLoader只有唯一的一个实例。 而Spring的单例作用域，是基于每个容器，每个Bean只有一个实例。这意味着，如果开发者根据一个类定义了一个Bean在单个的Spring容器中，那么Spring容器会根据Bean定义创建一个唯一的Bean实例。 单例作用域是*Spring*的默认作用域，下面的例子是在基于XML的配置中配置单例模式的Bean。

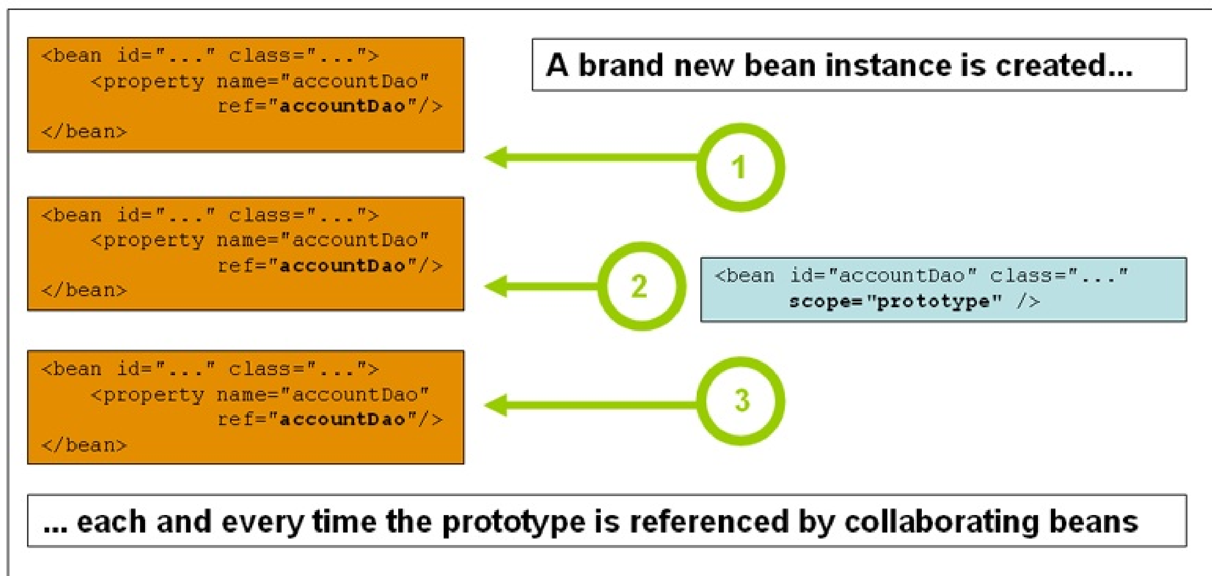
```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) --
>
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

The prototype scope

非单例的，原型的Bean指的就是每次请求Bean实例的时候，返回的都是新实例的Bean对象。也就是说，每次注入到另外的Bean或者通过调用 `getBean()` 来获得的Bean都将是全新的实例。 这是基于线程安全性的考虑，如果使用有状态的Bean对象用原型作用域，而无状态的Bean对象用单例作用域。

下面的例子说明了Spring的原型作用域。DAO通常不会配置为原型对象，因为典型的DAO是不会有任何的状态的。



下面的例子展示了XML中如何定义一个原型的Bean：

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

与其他的作用域相比，Spring是不会完全管理原型Bean的生命周期的：Spring容器只会初始化，配置以及装载这些Bean，传递给Client。但是之后就不会再去管原型Bean之后的动作了。 也就是说，初始化生命周期回调方法在所有作用域的Bean是都会调用的，但是销毁生命周期回调方法在原型Bean是不会调用的。所以，客户端代码必须注意清理原型Bean以及释放原型Bean所持有的一些资源。 可以通过使用自定义的 `bean post-processor` 来让Spring释放掉原型Bean所持有的资源。

在某些方面来说，Spring容器的角色就是取代了Java的 `new` 操作符，所有的生命周期的控制需要由客户端来处理。

Singleton beans with prototype-bean dependencies

当使用单例Bean的时候，而该单例Bean的依赖是原型Bean的时候，需要注意的是依赖的解析都是在初始化的阶段的。因此，如果将原型Bean注入到单例的Bean之中，只会请求一次原型的Bean，然后注入到单例的Bean之中。这个依赖的原型Bean仍然属于只有一个实例的。

然而，假设你需要单例Bean对原型的Bean的依赖需要每次在运行时都请求一个新的实例，那么你就不能够将一个原型的Bean来注入到一个单例的Bean当中了，因为依赖注入只会进行一次。 当Spring容器在实例化单例Bean的时候，就会解析以及注入它所需的依赖。如果实在需要每次都请求一个新的实例，可以参考Dependencies中的方法注入部分。

Request, session, global session, application, and WebSocket scopes

`request`，`session`以及`global session`这三个作用域都是只有在基于web的Spring `ApplicationContext`实现的（比如`XmlWebApplicationContext`）中才能使用。 如果开发者仅仅在常规的Spring IoC容器中比如`ClassPathXmlApplicationContext`中使用这些作用域，那么将会抛出一个`IllegalStateException`来说明使用了未知的作用域。

Initial web configuration

为了能够使用`request`，`session`以及`global session`作用域（web范围的Bean），需要在配置Bean之前配置做一些基础的配置。（对于标准的作用域，比如`singleton`以及`prototype`，是无需这些基础的配置的）

具体如何配置取决于Servlet的环境。

比如如果开发者使用了Spring Web MVC框架的话，每一个请求会通过Spring的`DispatcherServlet`或者`DispatcherPortlet`来处理的，也就没有其他特殊的初始化配置。`DispatcherServlet`和`DispatcherPortlet`已经包含了相关的状态。

如果使用Servlet 2.5的web容器，请求不是通过Spring的`DispatcherServlet`（比如JSF或者Struts）来处理。那么开发者需要注册`org.springframework.web.context.request.RequestContextListener`或者`ServletRequestListener`。 而在Servlet 3.0以后，这些都能够通过`WebApplicationInitializer`接口来实现。或者，如果是一些旧版本的容器的话，可以在`web.xml`中增加如下的Listener声明：

```
<web-app>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

如果是对Listener不甚熟悉，也可以考虑使用Spring的`RequestContextFilter`。Filter的映射取决于web应用的配置，开发者可以根据如下例子进行适当的修改。

```
<web-app>
    ...
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-clas
s>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

`DispatcherServlet` ，`RequestContextListener` 以及 `RequestContextFilter` 做的本质上完全一致，都是绑定request对象到服务请求的 `Thread` 上。这才使得Bean在之后的调用链上在请求和会话范围上可见。

Request scope

参考如下的Bean定义

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

Spring容器会在每次用到 `loginAction` 来处理每个HTTP请求的时候都会创建一个新的 `LoginAction` 实例。也就是说， `loginAction` Bean的作用域是HTTP `Request` 级别的。 开发者可以随意改变实例的状态，因为其他通过 `loginAction` 请求来创建的实例根本看不到开发者改变的实例状态，所有创建的Bean实例都是根据独立的请求来的。当请求处理完毕，这个Bean也会销毁。

Session scope

参考如下的Bean定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Spring容器会在每次调用到 `userPreferences` 在一个单独的HTTP会话周期来创建一个新的 `UserPreferences` 实例。换言之， `userPreferences` Bean的作用域是HTTP `Session` 级别的。 在 `request-scoped` 作用域的Bean上，开发者可以随意的更改实例的状态，同样，其他的HTTP `Session` 基本的实例在每个Session都会请求 `userPreferences` 来创建新的实例，所以开发者更改Bean的状态，对于其他的Bean仍然是不可见的。当HTTP `Session` 销毁了，那么根据这个 `Session` 来创建的Bean也就销毁了。

Global session scope

> 该部分主要是描述`portlet`的，详情可以Google更多关于`portlet`的相关信息。

参考如下的Bean定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

`global session`作用域比较类似之前提到的标准的HTTP `Session`，这种作用域是只应用于基于门户（portlet-based）的web应用的上下之中的。门户的Spec中定义的`global session`的意义：`global session`被所有构成门户的web应用所共享。定义为`global session`作用域的Bean是作用在全局门户`Session`的声明周期的。

如果在使用标准的基于Servlet的Web应用，而且定义了`global session`作用域的Bean，那么只是会使用标准的HTTP `Session`作用域，不会报错。

Application scope

考虑如下的Bean定义：

```
<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>
```

Spring容器会在整个web应用使用到`appPreferences`的时候创建一个新的`AppPreferences`的实例。也就是说，`appPreferences` Bean是在`ServletContext`级别的，好似一个普通的`ServletContext`属性一样。这种作用域在一些程度上来说和Spring的单例作用域是极为相似的，但是也有如下不同之处：

- `application`作用域是每个`ServletContext`中包含一个，而不是每个Spring `ApplicationContext`之中包含一个（某些应用中可能包含不止一个`ApplicationContext`）。
- `application`作用域仅仅作为`ServletContext`的属性可见，单例Bean是`ApplicationContext`可见。

Scoped beans as dependencies

Spring IoC容器不仅仅管理对象（Bean）的实例化，同时也负责装载依赖。如果开发者想装载一个Bean到一个作用域更广的Bean当中去（比如HTTP请求返回的Bean）,那么开发者选择注入一个AOP代理而不是短作用域的Bean。也就是说，开发者需要注入一个代理对象，这个代理对象既可以找到实际的Bean，也能够创建一个全新的Bean。

> 开发者会在单例Bean中使用 `<aop:scoped-proxy/>` 标签，来引用一个代理，这个代理的作用就是用来获取指定的Bean。 当声明使用 `<aop:scoped-proxy/>` 来生成一个原型Bean的时候，每个通过代理的调用都会产生一个新的目标实例。 并且，作用域代理并不是唯一来获取短作用域Bean的唯一安全的方式。开发者也可以通过简单的声明注入为 `ObjectFactory<MyTargetBean>`，允许通过类似 getObject() 之类的调用来获取一些指定的依赖，而不是直接储存依赖的实例。 `JSR-330`关于这部分的名称不同叫做 Provider ，通过使用 Provider声明和一个相关的 `get()` 方法来获取指定的依赖。详细关于JSR-330的信息可以进去详细了解。

请参考下面的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/>
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.foo.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

使用代理，只需要在短作用域的Bean定义之中加入一个子节点 `<aop:scoped-proxy/>` 即可。Dependencies中的方法注入中就提及到了Bean依赖的一些问题，这也是我们为什么要使用 `aop` 代理的原因。假设我们没有使用 `aop` 代理而是直接进行依赖注入，参考如下的例子：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

上面的例子中， `userManager` 明显是一个单例的Bean，注入了一个HTTP `Session` 级别的 `userPreferences` 依赖，显然的问题就是 `userManager` 在Spring容器中只会实例化一次，而依赖(当前例子中的 `userPreferences`)也只能注入一次。这也就意味着 `userManager` 每次使用的都是相同的 `userPreferences` 对象。

那么这种情况就绝对不是开发者想要的那种将短作用域注入到长作用域Bean中的情况了，举例来说，注入一个HTTP `Session` 级别的Bean到一个单例之中，或者说，当开发者通过 `userManager` 来获取指定与某个HTTP `Session` 的 `userPreferences` 对象都是不可能的。所以容器创建了一个获取 `UserPreferences` 对象的接口，这个接口可以根据Bean对象作用域机制来获取与作用域相关的对象（比如说HTTP `Request` 或者HTTP `Session` 等）。容器之后注入代理对象到 `userManager` 中，而意识不到所引用 `UserPreferences` 是代理。在这个例子之中，当 `UserManager` 实例调用方法来获取注入的依赖 `UserPreferences` 对象时，其实只会调用了代理的方法，由代理去获取真正的对象，在这个例子中就是HTTP `Session` 级别的Bean。

所以当开发者希望能够正确的使用配置 `request`，`session` 或者 `globalSession` 级别的Bean来作为依赖时，需要进行如下的类似配置：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

选择代理的类型

默认情况下，Spring容器创建代理的时候标记为 `<aop:scoped-proxy/>` 的标签时，会创建一个基于*CGLIB*的代理。

> CGLIB代理会拦截 `public` 方法调用！所以不要在非 `public` 方法上使用代理，这样将不会获取到指定的依赖。

或者，开发者可以通过指 `<aop:scoped-proxy/>` 标签的 `proxy-target-class` 属性的值为 `false` 来配置Spring容器来为这些短作用域的Bean创建一个标准JDK的基于接口的代理。使用JDK基于接口的代理意味着开发者不需要在应用的路径引用额外的库来完成代理。当然，这也意味着短作用域的Bean需要额外实现一个接口，而依赖是从这些接口来获取的。

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

`DefaultUserPreferences` 实现了 `UserPreferences` 而且提供了接口来获取实际的对象。更多的信息可以参考AOP代理。

Custom scopes

Bean的作用域机制是可扩展的，开发者可以定义自己的一些作用域，甚至重新定义已经存在的作用域，但是这一点Spring团队是不推荐的，并且开发者不能够重写`singleton`以及`prototype`作用域。

Creating a custom scope

为了能够使Spring可以管理开发者定义的作用域，开发者需要实现`org.springframework.beans.factory.config.Scope`接口。想知道如何实现开发者自己定义的作用域，可以参考Spring框架的一些实现或者是`Scope`的javadoc，里面会解释开发者需要实现的一些细节。

`Scope`接口中含有4个方法来获取对象，移除对象，允许销毁等。

下面的方法返回一个存在的作用域的对象。比如说`Session`的作用域实现，该函数将返回会话作用域的Bean（如果Bean不存在，该方法会创建一个新的实例）

```
Object get(String name, ObjectFactory objectFactory)
```

下面的方法会将对象移出作用域。同样，以`Session`为例，该函数会删除`Session`作用域的Bean。删除的对象会作为返回值返回，当无法找到对象的时候可以返回`null`。

```
Object remove(String name)
```

下面的方法会注册一个回调方法，当需要销毁或者作用域销毁的时候调用。详细可以参考在javadoc和Spring作用域的实现中找到更多关于销毁回调方法的信息。

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

下面的方法会获取作用域的区分标识，区分标识区别于其他的作用域。

```
String getConversationId()
```

Using a custom scope

在实现了开发者的自定义作用域之后，开发者还需要让Spring容器能够识别发现这个新的作用域。下面的方法就是在Spring容器中用来注册新的作用域的。

```
void registerScope(String scopeName, Scope scope);
```

这个方法是在`ConfigurableBeanFactory`的接口中声明的，在大多数的`ApplicationContext`的实现中都是可以用的，可以通过`BeanFactory`属性来调用。

`registerScope(..)`方法的第一个参数是作用域相关联的唯一的一个名字；举例来说，比如Spring容器之中的`singleton`和`prototype`就是这样的名字。第二个参数就是我们根据Scope接口所实现的具体的对象。

假定开发者实现了自定义的作用域，然后按照如下步骤来注册。

> 下面的例子使用了`SimpleThreadScope`，这个例子Spring中是有实现的，但是没有默认注册。开发者自实现的`Scope`也可以通过如下方式来注册。

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

之后，开发者可以通过如下类似的Bean定义来使用自定义的`Scope`：

```
<bean id="..." class="..." scope="thread">
```

在定制的`Scope`中，开发者也不限于仅仅通过编程方式来注册自己的`Scope`，开发者可以通过下面`CustomScopeConfigurer`类来实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="org.springframework.context.support.SimpleThreadScope
"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>

</beans>
```

Customing the nature of a bean

Lifecycle callbacks

开发者通过实现Spring的 `InitializeingBean` 和 `DisposableBean` 接口，就可以让容器来管理Bean的生命周期。容器会在调用 `afterPropertiesSet()` 之后和 `destroy()` 之前会允许Bean在初始化和销毁Bean的时候执行一些操作。

> JSR-250的 `@PostConstruct` 和 `@PreDestroy` 注解就是现代Spring应用生命周期回调的最佳实践。使用这些注解意味着Bean不会再耦合在Spring特定的接口上。详细内容，后续将会介绍。 如果开发者不想使用JSR-250的注解，仍然可以考虑使用 `init-method` 和 `destroy-method的` 定义来解耦Spring接口。

内部来说，Spring框架使用 `BeanPostProcessor` 的实现来处理接口的回调， `BeanPostProcessor` 能够找到并调用合适的方法。如果开发者需要定制一些Spring并不直接提供的生命周期行为，开发者可以考虑自行实现一个 `BeanPostProcessor` 。更多的信息可以参考后面的容器扩展点。

除了初始化和销毁回调，Spring管理的对象也实现了 `Lifecycle` 接口来让管理的对象在容器的生命周期内启动和关闭。

生命周期回调在本节会进行详细描述。

Initialization callbacks

`org.springframework.beans.factory.InitializingBean` 接口允许Bean在所有的必要的依赖配置配置完成后来执行初始化Bean的操作。 `InitializingBean` 接口中特指了一个方法：

```
void afterPropertiesSet() throws Exception;
```

Spring团队是建议开发者不要使用 `InitializingBean` 接口的，因为这样会将代码耦合到Spring的特定接口之上。而通过使用 `@PostConstruct` 注解或者指定一个POJO的实现方法，会比实现接口要更好。在基于XML的配置元数据上，开发者可以使用 `init-method` 属性来指定一个没有参数的方法。使用Java配置的开发者可以使用 `@Bean` 之中的 `initMethod` 属性，比如如下：

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {

    public void init() {
        // do some initialization work
    }

}
```

与如下代码一样效果：

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
    }

}
```

但是前一个版本的代码是没有耦合到Spring的。

Destruction callbacks

实现了 `org.springframework.beans.factory.DisposableBean` 接口的Bean就能通让容器通过回调来销毁Bean所引用的资源。 `DisposableBean` 接口包含了一个方法：

```
void destroy() throws Exception;
```

同InitializingBean相类似，Spring团队仍然不建议开发者来实现 `DisposableBean` 回调接口，因为这样会将开发者的代码耦合到Spring代码上。换种方式，比如使用 `@PreDestroy` 注解或者指定一个Bean支持的配置方法，比如在基于XML的配置元数据中，开发者可以在Bean标签上指定 `destroy-method` 属性。而在基于Java配置中，开发者也可以配置 `@Bean` 的 `destroyMethod` 来实现销毁回调。

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

上面的代码配置和如下配置是等同的：

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }

}
```

但是第一段代码是没有耦合到Spring的。

> `<bean/>` 标签的 `destroy-method` 可以被配置为特殊指定的值，来方便让Spring能够自动的检查到 `close` 或者 `shutdown` 方法（可以实现 `java.lang.AutoCloseable` 或者 `java.io.Closeable` 都会匹配。）这个特殊指定的值可以配置到 `<beans/>` 的 `default-destroy-method` 来让所有的Bean实现这个行为。

Default initialization and destroy methods

当开发者不使用Spring特有的 `InitializingBean` 和 `DisposableBean` 回调接口来实现初始化和销毁方法的时候，开发者通常定义的方法名字都是好似 `init()` ， `initialize()` 或者是 `dispose()` 等等。那么，可以将这类的方法在项目中标准化，来让所有的开发者都使用一样的名字来确保一致性。

开发者可以配置Spring容器来针对每一个Bean都查找这种名字的初始化和销毁回调函数。也就是说，任何的一个应用开发者，都会在应用的类中使用一个叫 `init()` 的初始化回调，而不需要在每个Bean中定义 `init-method="init"` 这中属性。Spring IoC容器会在Bean创建的时候调用那个方法（就如前面描述的标准生命周期一样。）这个特性也将强制开发者为其他的初始化以及销毁回调函数使用同样的名字。

假设开发者的初始化回调方法名字为 `init()` 而销毁的回调方法为 `destroy()` 。那么开发者的类就会好似如下的代码：

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }

}
```

```
<beans default-init-method="init">

    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

`<beans/>` 标签上面的 `default-init-method` 属性会让Spring IoC容器识别叫做 `init` 的方法来作为Bean的初始化回调方法。当Bean创建和装载之后，如果Bean有这么一个方法的话，Spring容器就会在合适的时候调用。

类似的，开发者也可以配置默认销毁回调函数，基于XML的配置就在 `<beans/>` 标签上面使用 `default-destroy-method` 属性。

当存在一些Bean的类有了一些回调函数，而和配置的默认回调函数不同的时候，开发者可以通过特指的方式来覆盖掉默认的回调函数。以XML为例，就是通过使用 `<bean>` 标签的 `init-method` 和 `destroy-method` 来覆盖掉 `<beans/>` 中的配置。

Spring容器会做出如下保证，Bean会在装载了所有的依赖以后，立刻就开始执行初始化回调。这样的话，初始化回调只会在直接的Bean引用装载好后调用，而AOP拦截器还没有应用到Bean上。首先目标Bean会完全初始化好，然后，AOP代理以及其拦截链才能应用。如果目标Bean以及代理是分开定义的，那么开发者的代码甚至可以跳过AOP而直接和引用的Bean交互。因此，在初始化方法中应用拦截器会前后矛盾，因为这样做耦合了目标Bean的生命周期和代理/拦截器，还会因为同Bean直接交互而产生奇怪的现象。

Combining lifecycle mechanisms

在Spring 2.5之后，开发者有三种选择来控制Bean的生命周期行为：

- `InitializingBean`和`DisposableBean`回调接口
- 自定义的`init()`以及`destroy`方法
- 使用`@PostConstruct`以及`@PreDestroy`注解

开发者也可以在Bean上联合这些机制一起使用

> 如果Bean配置了多个生命周期机制，而且每个机制配置了不同的方法名字，那么每个配置的方法会按照后面描述的顺序来执行。然而，如果配置了相同的名字，比如说初始化回调为`init()`，在不止一个生命周期机制配置为这个方法的情况下，这个方法只会执行一次。

如果一个Bean配置了多个生命周期机制，并且含有不同的方法名，执行的顺序如下：

- 包含`@PostConstruct`注解的方法
- 在`InitializingBean`接口中的`afterPropertiesSet()`方法
- 自定义的`init()`方法

销毁方法的执行顺序和初始化的执行顺序相同：

- 包含`@PreDestroy`注解的方法
- 在`DisposableBean`接口中的`destroy()`方法
- 自定义的`destroy()`方法

Startup and shutdown callbacks

`Lifecycle`接口中为任何有自己生命周期需求的对象定义了一些基本的方法（比如启动和停止一些后台进程）：

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();

}
```

任何Spring管理的对象都可实现上面的接口。那么当`ApplicationContext`本身收到了启动或者停止的信号时，比如运行时的停止或者重启等场景，`ApplicationContext`会通知到所有上下文中包含的生命周期对象，`ApplicationContext`通过将代理到`LifecycleProcessor`来串联上下文中的`Lifecycle`的实现对象。

```
public interface LifecycleProcessor extends Lifecycle {

    void onRefresh();

    void onClose();

}
```

从上面代码我们可以发现`LifecycleProcessor`是`Lifecycle`接口的扩展。`LifecycleProcessor`增加了另外的两个方法来针对上下文的刷新和关闭做出反应。

> 常规的`org.springframework.context.Lifecycle`接口只是为明确的开始/停止通知提供一个契约，而并不表示在上下文刷新会自动开始。考虑实现`org.springframework.context.SmartLifecycle`接口则可以取代在某个Bean的自动启动过程（包括启动阶段）。同时，停止通知并不能保证在销毁之前出现：在正常的关闭情况下，所有的`Lifecycle` Bean都会在销毁回调准备好之前收到停止通知，然而，在上下文存活期的热刷新或者停止刷新尝试的时候，只会调用销毁方法。

启动和关闭调用是很重要的。如果不同的Bean之间存在`depends-on`的关系的话，被依赖的一方需要更早的启动，而且关闭的更早。然而，有的时候直接的依赖是未知的，而开发者仅仅知道哪一种类型需要更早进行初始化。在这种情况下，`SmartLifecycle`接口定义了另一种选项，就是其父接口`Phased`中的`getPhase()`方法。

```
public interface Phased {

    int getPhase();

}
```

```
public interface SmartLifecycle extends Lifecycle, Phased {

    boolean isAutoStartup();

    void stop(Runnable callback);

}
```

当启动时，拥有最低的`phased`的对象优先启动，而当关闭时，是相反的顺序。因此，如果一个对象实现了`SmartLifecycle`然后令其`getPhase()`方法返回了`Integer.MIN_VALUE`的话，就会让该对象最早启动，而最晚销毁。显然，如果`getPhase()`方法返回了`Integer.MAX_VALUE`就说明了该对象会最晚启动，而最早销毁。当考虑到使用`phased`的值得时候，也同时需要了解正常没有实现`SmartLifecycle`的`Lifecycle`对象的默认值，这个值为0。因此，任何负值将标明对象会在标准组件启动之前启动，在标准组件销毁以后再进行销毁。

`SmartLifecycle` 接口也定义了一个 `stop` 的回调函数。任何实现了 `SmartLifecycle` 接口的函数都必须在关闭流程完成之后调用回调中的 `run()` 方法。这样做可以使能异步关闭。而 `LifecycleProcessor` 的默认实现 `DefaultLifecycleProcessor` 会等到配置的超时时间之后再调用回调。默认的每一阶段的超时时间为30秒。开发者可以通过定义一个叫做 `lifecycleProcessor` 的Bean来覆盖默认的生命周期处理器。如果开发者需要配置超时时间，可以通过如下代码进行配置：

```
<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">
    <!-- timeout value in milliseconds -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

前文提到的， `LifecycleProcessor` 接口定义了回调方法来刷新和关闭上下文。关闭的话，如果 `stop()` 方法已经明确调用了，那么就会驱动关闭的流程，但是如果是上下文正在关闭就不会发生这种情况。而刷新的回调会使用 `SmartLifecycle` 的另一个特性。当上下文刷新完毕（所有的对象已经实例化并初始化），那么就会调用回调，默认的生命周期处理器会检查每一个 `SmartLifecycle` 对象的 `isAutoStartup()` 返回的Bool值。如果为真，对象将会自动启动而不是等待明确的上下文调用，或者调用自己的 `start()` 方法(不同于上下文刷新，标准的上下文实现是不会自动启动的)。 `phased` 的值以及 `depends-on` 关系会决定对象启动和销毁的顺序。

Shutting down the Spring IoC container gracefully in non-web applications

> 这一部分只是针对于非Web的应用。Spring的基于web的 `ApplicationContext` 实现已经有代码在web应用关闭的时候能够自动的的关闭Spring IoC容器。

如果开发者在非web应用环境使用Spring IoC容器的话， 比如，在桌面客户端的环境下，开发者需要在JVM上注册一个关闭的钩子，来确保在关闭Spring IoC容器的时候能够调用相关的销毁方法来释放掉引用的资源。当然，开发者也必须要正确的配置和实现那些销毁回调。

开发者可以在 `ConfigurableApplicationContext` 接口调用 `registerShutdownHook()` 来注册销毁的钩子：

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext(
                new String []{"beans.xml"});

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...

    }
}
```

ApplicationContextAware and BeanNameAware

当 `ApplicationContext` 在创建实现了 `org.springframework.context.ApplicationContextAware` 接口的对象时，该对象的实例会包含一个到 `ApplicationContext` 的引用。

```
public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext applicationContext) throws BeansExce
ption;

}
```

这样Bean就能够通过编程的方式操作和创建 `ApplicationContext` 了。通过 `ApplicationContext` 接口，或者通过将引用转换成已知的接口的子类，比如 `ConfigurableApplicationContext` 就能够提供一些额外的功能。其中的一个用法就是可以通过编程的方式来获取其他的Bean。有的时候这个能力很有用。当然，Spring团队推荐最好不要这样做，因为这样会耦合代码到Spring上，同时也没有遵循IoC的风格。 `ApplicationContext` 的其它的方法可以提供一些到诸如资源的访问，发布应用事件，或者进入 `MessageSource` 之类的功能。这些信息在后续的针对 `ApplicationContext` 的描述中会讲到。

在Spring 2.5版本中，自动装载也是获得 `ApplicationContext` 的一种方式。传统的构造函数和通过类型的装载方式（前文依赖中有相关描述）可以通过构造函数或者是setter方法的方式注入，开发者也可以通过注解注入的方式。

当`ApplicationContext`创建了一个实现了`org.springframework.beans.factory.BeanNameAware`接口的类，那么这个类就可以针对其名字进行配置。

```
public interface BeanNameAware {

    void setBeanName(string name) throws BeansException;

}
```

这个回调的调用处于属性配置完以后，但是初始化回调之前。比如`InitializingBean`的`afterPropertiesSet()`方法以及自定义的初始化方法等。

Other Aware interfaces

除了上面描述的两种Aware接口，Spring还提供了一系列的`Aware`接口来让Bean告诉容器，这些Bean需要一些具体的基础设施信息。最重要的一些`Aware`接口都在下面表中进行了描述：

名字	注入的依赖
`ApplicationContextAware`	声明的 `ApplicationContext`
`ApplicationEventPlulisherAware`	`ApplicationContext` 中的事件发布器
`BeanClassLoaderAware`	加载Bean使用的类加载器
`BeanFactoryAware`	声明的 `BeanFactory`
`BeanNameAware`	Bean的名字
`BootstrapContextAware`	容器运行的资源适配器 `BootstrapContext` ，通常仅在JCA环境下有效
`LoadTimeWeaverAware`	加载期间处理类定义的weaver
`MessageSourceAware`	解析消息的配置策略
`NotificationPublisherAware`	Spring JMX通知发布器
`PortletConfigAware`	容器当前运行的 `PortletConfig` ，仅在web下的Spring `ApplicationContext` 中可见
`PortletContextAware`	容器当前运行的 `PortletContext` ，仅在web下的Spring `ApplicationContext` 中可见
`ResourceLoaderAware`	配置的资源加载器
`ServletConfigAware`	容器当前运行的 `ServletConfig` ，仅在web下的Spring `ApplicationContext` 中可见
`ServletContextAware`	容器当前运行的 `ServletContext` ，仅在web下的Spring `ApplicationContext` 中可见

再次的声明，上面这些接口的使用是违反IoC原则的，除非必要，最好不要使用。

Bean definition inheritance

bean的定义可以包含很多配置信息，包括构造方法参数，属性值和容器特定的信息，如初始化方法，静态工厂方法名称等。子bean定义从继承父bean定义的配置元数据。子bean可以覆盖或者添加一些它所需要的值。使用父子bean定义可以节省很多配置输入。实际上，这是一种模板形式。

如果你编程式地使用 `ApplicationContext` 接口，子bean的定义可以通过 `ChildBeanDefinition` 类来表示。很多用户不使用这个级别的方法，而是在类似于 `ClassPathXmlApplicationContext` 中声明式地配置bean的定义。当你使用基于XML配置时，你可以在子bean中用 `parent` 属性，该属性的值用来标识父bean。

```
<bean id="inheritedTestBean" abstract="true"
        class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
        class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBean" init-method="initialize">
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>
```

子bean如果没有指定class，它将使用父bean定义的class，但也可以进行重载。在后一种情况中，子bean必须与父bean兼容，也就是说，它必须接受父bean的属性值。

子bean定义从父类继承作用域，构造器参数，属性值，和可以重写的方法，除此之外，还可以增加新的值。你指定的任何作用域，初始化方法，销毁方法，和/或者静态工厂方法设置，都会覆盖相应的父bean设置。

其余的设置总是取自子bean定义：depends on, autowire mode, dependency check, singleton, scope, lazy init。

上面的例子，通过使用 `abstract` 属性明确地表明这个父bean定义是抽象的。如果，父bean定义没有明确地指出所属的类，那么标记父bean定义为 `abstract` 是必须的，如下：

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBeanWithoutClass" init-method="initialize">
    <property name="name" value="override"/>
    <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

这个父bean不能自主实例化，因为它是不完整的，同时它也明确地被标注为 `abstract` ；像这样， 一个bean定义为 `abstract` 的，它只能作为一个纯粹的bean模板，为子bean定义，充当父bean定义。尝试独立地使用这样一个 `abstract` 的父bean，把他作为另一个bean的引用，或者根据这个父bean的id显式调用 `getBean()` 方法，将会返回一个错误。类似地，容器内部的 `preInstantiateSingletons()` 方法，也忽略定义为抽象的bean定义。

> `ApplicationContext`默认会预实例化所有单例bean。因此，如果你打算把一个(父)bean定义仅仅作为模板来使用，同时给它指定了`class`属性，你必须确保设置`abstract`属性为true,否则，应用程序上下文，会(尝试)预实例化这个`abstract`bean。

6.8 Container Extension Points

通常，应用程序开发者，不需要继承 `ApplicationContext` 的实现类。相反，Spring IoC容器可以通过插入特殊的集成接口的实现进行拓展。新的一节中，描述了这些集成接口。

6.8.1 Customizing beans using a BeanPostProcessor

`BeanPostProcessor` 定义了回调方法，通过实现这个回调方法，你可以提供你自己的(或者重写容器默认的)实例化逻辑，依赖分析逻辑等等。如果你想在Spring容器完成实例化，配置，和初始化bean之后，实例化一些自定义的逻辑，你可以插入一个或多个 `BeanPostProcessor` 的实现。

你可以配置多个 `BeanPostProcessor` 实例，你可以通过设置 `order` 属性来控制这些BeanPostProcessors执行的顺序。你可以设置这个属性仅当 `BeanPostProcessor` 实现了 `Ordered` 接口。如果你编写自己的 `BeanPostProcessor` 你也应该考虑实现 `Ordered` 接口。更多细节，请查看 `BeanPostProcessor` 和 `Ordered` 接口的javadocs，也可以看看下面的要点programmatic registration of `BeanPostProcessors` 。

> `BeanPostProcessors` 作用在一个bean(或者对象)的实例上;也就是说，Spring IoC实例化一个bean实例之后， BeanPostProcessors，才开始进行处理。 `BeanPostProcessors` 作用范围是每一个容器。这仅仅和你正在使用容器有关。如果你在一个容器中定义了一个 `BeanPostProcessor` ，它将仅仅后置处理那个容器中的beans。换言之，一个容器中的beans不会被另一个容器中的 `BeanPostProcessor` 处理，即使这两个容器，具有相同的父类。 为了改变实际的bean定义(例如， blueprint 定义的bean)，你反而需要使用 `BeanFactoryPostProcessor` ，就像在Section 5.8.2, “Customizing configuration metadata with a BeanFactoryPostProcessor”中描述的那样。

`org.springframework.beans.factory.config.BeanPostProcessor` 接口，由两个回调方法组成。当这样的一个类注册为容器的一个后置处理器，由于每一个bean实例都是由容器创建的，这个后置处理器会在容器的初始化方法(比如InitializingBean的afterPropertiesSet()和任何生命的初始化方法)被调用之前和任何bean实例化回调之后从容器得到一个回调方法。后置处理器，可以对bean采取任何措施，包括完全忽略回调。一个bean后置处理器，通常会检查回调接口或者使用代理包装一个bean。一些Spring AOP基础设施类，为了提供包装式的代理逻辑，被实现为bean后置处理器。

`ApplicationContext` 会自动地检测所有定义在配置元文件中，并实现了 `BeanPostProcessor` 接口的bean。该 `ApplicationContext` 注册这些beans作为后置处理器，使他们可以在bean创建完成之后被调用。bean后置处理器可以像其他bean一样部署到容器中。

注意，当在一个配置类上，使用@Bean工厂方法声明一个 `BeanPostProcessor` ，工厂方法返回的类型应该是实现类自身，或至少也要是 `org.springframework.beans.factory.config.BeanPostProcessor` 接口，要清楚地表明这个bean的后置处理器本质特点。否则，在它完全创建之前， `ApplicationContext` 将不能通过类型自动探测它。由于一个BeanPostProcessor，早期就需要被实例化，以适应上下文中其他bean的实例化，因此这个早期的类型检查是至关重要的。

> 虽然推荐使用 `ApplicationContext` 的自动检测来注册 `BeanPostProcessor` ，但是也可以编程式地使用 `ConfigurableBeanFactory` 的 `addBeanPostProcessor` 方法来注册。这对于在注册之前需要对条件逻辑进行评估，或者是在继承层次的上下文之间复制bean后置处理器是有很有用的。但是请注意，编程地添加的 `BeanPostProcessors` 不需要考虑 `Ordered` 接口。也就是注册的顺序决定了执行的顺序。也要注意，编程式注册的 `BeanPostProcessors` ，总是预先被处理----早于通过自动检测方式注册的，同时忽略任何明确的排序。
>
> 实现了 `BeanPostProcessor` 接口的类是特殊的,会被容器特殊处理。所有 `BeanPostProcessors` 和他们直接引用的 beans都会在容器启动的时候被实例化,作为 `ApplicationContext` 特殊启动阶段的一部分。接着，所有的 `BeanPostProcessors` 以一个有序的方式进行注册，并应用于容器中的一切bean。因为AOP自动代理本身被实现为 `BeanPostProcessor` ，这个 `BeanPostProcessors` 和它直接应用的beans都没有资格进行自动代理，这样就没有切面编织到他们里面。对于所有这样的bean，你会看到一个info日志："Bean foo is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)"。
>
> 注意，如果你有beans使用自动装配或者 `@Resource` 装配到了你的 `BeanPostProcessor` 中，当根据依赖搜索匹配类型时，Spring也许会访问意外类型的bean；因此，使它们没有资格进行自动代理，或者其他类型的bean后置处理。例如，你使用 `@Resource` 注解一个依赖，其中字段或者set方法名，不是和bean声明的名字直接对应，同时没有name属性被使用，然后，Spring将会根据类型，访问其他beans进行匹配。

下面的示例显示了如何在 `ApplicationContext` 中编写，注册，使用 `BeanPostProcessor` 。

Example: Hello World, BeanPostProcessor-style

第一个示例演示了基础的使用。示例中演示了一个自定义的 `BeanPostProcessor` 实现，在容器创建bean后调用了每个bean的 `toString()` 方法，并把结果输出到控制台上。

以下是自定义 `BeanPostProcessor` 实现类的定义：

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean,
            String beanName) throws BeansException {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean,
            String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang.xsd">

    <lang:groovy id="messenger"
            script-source="classpath:org/springframework/scripting/groovy/Messenger.gr
oovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
    when the above bean (messenger) is instantiated, this custom
    BeanPostProcessor implementation will output the fact to the system console
    -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

注意 `InstantiationTracingBeanPostProcessor` 是简单地定义。它甚至没有名字，因为它能像其它bean一样被依赖注入。（上面示例中也定义了一个使用Groovy脚本支持的bean。Spring动态语言支持详见Chapter 34, Dynamic language support）

下面简单的Java应用执行了前面代码和配置：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.x
ml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger);
    }

}
```

上面应用运行输出结果如下： Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961 org.springframework.scripting.groovy.GroovyMessenger@272961

Example: The RequiredAnnotationBeanPostProcessor

自定义 `BeanPostProcessor` 实现与回调接口或注解配合使用，是一种常见的扩展Spring IoC容器的手段。一个例子就是 `RequiredAnnotationBeanPostProcessor` -这是一个 `BeanPostProcessor` 实现，确保用（任意）注解标记的那些JavaBean属性确实被注入一个值。

6.8.2 Customizing configuration metadata with a BeanFactoryPostProcessor

下一个我们要看的扩展点是 `org.springframework.beans.factory.config.BeanFactoryPostProcessor` 。这个接口的语义与 `BeanPostProcessor` 类似，但有一个主要的不同点： `BeanFactoryPostProcessor` 操作bean的配置元数据；也就是说，Spring的IoC容器允许 `BeanFactoryPostProcessor` 来读取配置元数据并在容器实例化任何bean(除了 `BeanFactoryPostProcessor`)之前可以修改它。

你可以配置多个 `BeanFactoryPostProcessor` 实例，你可以通过设置 `order` 属性来控制这些 `BeanFactoryPostProcessor` 执行的顺序。你可以设置这个属性仅当 `BeanFactoryPostProcessor` 实现了 `Ordered` 接口。如果你编写自己的 `BeanFactoryPostProcessor` 你也应该考虑实现 `Ordered` 接口。更多细节，请查看 `BeanPostProcessor` 和 `Ordered` 接口的javadocs。

> 如果你想修改真实的bean实例（也就是说，从配置元数据中创建的对象），那么你需要使用`BeanPostProcessor`（在上面 6.8.1 节，“Customizing beans using a BeanPostProcessor”中描述）来代替。在`BeanFactoryPostProcessor`（比如使用BeanFactory.getBean()）中来使用这些bean的实例虽然在技术上是可行的，但这么来做会引起bean过早实例化，违反标准的容器生命周期。这也会引发一些副作用，比如绕过bean的后置处理。
>
> `BeanFactoryPostProcessors`作用范围是每一个容器。这仅仅和你正在使用容器有关。如果你在一个容器中定义了一个`BeanFactoryPostProcessor`，它将仅仅后置处理那个容器中的beans。换言之，一个容器中的beans不会被另一个容器中的`BeanFactoryPostProcessor`处理，即使这两个容器，具有相同的父类。

当在`ApplicationContext`中声明时，bean工厂后置处理器会自动被执行，这就可以对定义在容器中的配置元数据进行修改。Spring包含了一些预定义的bean工厂后置处理器，比如`PropertyOverrideConfigurer`和`PropertyPlaceholderConfigurer`。自定义的`BeanFactoryPostProcessor`也可以用来，比如，注册自定义的属性编辑器。

`ApplicationContext`会自动检测任意部署其中，且实现了`BeanFactoryPostProcessor`接口的bean。在适当的时间，它用这些bean作为bean工厂后置处理器。你可以部署这些后置处理器bean作为你想用的任意其它的bean。

> 和`BeanPostProcessor`一样，通常你不会想配置`BeanFactoryPostProcessor`来进行延迟初始化。如果没有其它bean引用`Bean(Factory)PostProcessor`，那么后置处理器就不会被初始化了。因此，标记它为延迟初始化就会被忽略，即便你在`<beans/>`元素声明中设置`default-lazy-init`属性为`true`，那么`Bean(Factory)PostProcessor`也会正常被初始化。

Example: the Class name substitution PropertyPlaceholderConfigurer

从使用了标准Java `Properties`格式的bean定义的分离的文件中，你可以使用`PropertyPlaceholderConfigurer`来具体化属性值。这么做允许部署应用程序来自定义指定的环境属性，比如数据库的连接URL和密码，不会有修改容器的主XML定义文件或其它文件的复杂性和风险。

考虑一下下面这个基于 XML 的配置元数据代码片段，这里的`DataSource`就使用了占位符来定义。这个示例展示了从`Properties`文件中配置属性的方法。在运行时，`PropertyPlaceholderConfigurer`就会用于元数据并为数据源替换一些属性。指定替换的值作为`${property-name}`形式中的占位符，这里应用了 Ant/log4j/JSP EL 的风格。

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>

<bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

而真正的值是来自于标准的 Java Properties 格式的文件：

> jdbc.driverClassName=org.hsqldb.jdbcDriver
> jdbc.url=jdbc:hsqldb:hsql://production:9002
> jdbc.username=sa
> jdbc.password=root

因此，字符串${jdbc.username}在运行时会被值'sa'替换，对于其它占位符来说也是相同的，匹配到了属性文件中的键就会用其值替换占位符。 `PropertyPlaceholderConfigurer` 在bean定义的属性中检查占位符。此外，对占位符可以自定义前缀和后缀。

使用 Spring 2.5 引入的 `context` 命名空间，也可以使用专用的配置元素来配置属性占位符。在 `location` 属性中，可以提供一个或多个以逗号分隔的列表。

```
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

`PropertyPlaceholderConfigurer` 不仅仅查看在 `Properties` 文件中指定的属性。默认情况下，如果它不能在指定的属性文件中找到属性，它也会检查Java `System` 属性。 你可以通过设置 `systemPropertiesMode` 属性，使用下面整数的三者之一来自定义这种行为：

- never(0)：从不检查系统属性
- fallback(1)：如果没有在指定的属性文件中解析到属性，那么就检查系统属性。这是默认的情况。
- override(2)：在检查指定的属性文件之前，首先去检查系统属性。这就允许系统属性覆盖其它任意的属性资源。

查看 `PropertyPlaceholderConfigurer` 的JavaDoc文档来获取更多信息。

> 你可以使用 `PropertyPlaceholderConfigurer` 来替换类名，在运行时，当你不得不去选择一个特定的实现类时，这是很有用的。比如：

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <value>classpath:com/foo/strategy.properties</value>
    </property>
    <property name="properties">
        <value>custom.strategy.class=com.foo.DefaultStrategy</value>
    </property>
</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

如果类在运行时不能解析成一个有效的类，那么在即将创建时，bean 的解析就失败了，这是 `ApplicationContext` 在对非延迟初始化bean的 `preInstantiateSingletons()` 阶段发生的。

Example: the PropertyOverrideConfigurer

`PropertyOverrideConfigurer` ，另外一种bean工厂后置处理器，类似于 `PropertyPlaceholderConfigurer` ，但不像后者，对于所有bean的属性，原始定义可以有默认值或没有值。如果一个 `Properties` 覆盖文件没有特定bean的属性配置项，那么就会使用默认的上下文定义。

注意，bean定义是不知道被覆盖的，所以从XML定义文件中不能立即明显反应覆盖配置被使用中。在多个 `PropertyOverrideConfigurer` 实例的情况下，为相同 bean 的属性定义不同的值，那么最后一个有效，源于它的覆盖机制。

属性文件配置行像这种格式：

```
beanName.property=value
```

例如：

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

这个示例文件可以用于包含了dataSource bean的容器，它有driver和url属性。

复合属性名也是支持的，除了最终的属性被覆盖，只要路径中的每个组件都是非空的（假设由构造方法初始化）。在这个例子中...

```
foo.fred.bob.sammy=123
```

`foo` bean的 `fred` 属性的 `bob` 属性的 `sammy` 属性的值设置为标量123。

> 指定的覆盖值通常是文字值；它们不会被转换成bean的引用。当XML中的bean定义的原始值指定了 bean 引用时，这个约定也适用。

使用 Spring 2.5 引入的 `context` 命名空间，可以使用专用的配置元素来配置属性覆盖：

```
<context:property-override location="classpath:override.properties"/>
```

6.8.3 Customizing instantiation logic with a FactoryBean

实现了 `org.springframework.beans.factory.FactoryBean` 接口的对象是它们自己的工厂。

`FactoryBean` 接口就是Spring IoC容器实例化逻辑的可插拔点。如果你的初始化代码很复杂，那么相对于（潜在地）大量详细的 XML 而言，最好是使用 Java 语言来表达。你可以创建自己的 `FactoryBean` ，在类中编写复杂的初始化代码，之后将你自定义的 `FactoryBean` 插入到容器中。 `FactoryBean` 接口提供下面三个方法：

- `Object getObject()` ：返回工厂创建对象的实例。这个实例可能被共享，那就是看这个工厂返回的是单例还是原型实例了。
- `boolean isSingleton()` ：如果 `FactoryBean` 返回单例的实例，那么该方法返回 `true` ，否则就返回 `false` 。
- `Class getObjectType()` ：返回由 `getObject()` 方法返回的对象类型，或者事先不知道类型时返回 `null` 。

`FactoryBean` 的概念和接口被用于 Spring Framework 中的很多地方；随 Spring 发行，有超过50个 `FactoryBean` 接口的实现类。

当你需要向容器请求一个真实的 `FactoryBean` 实例，而不是它生产的 bean，调用 `ApplicationContext` 的 `getBean()` 方法时在bean的id之前加连字符 `（&）` 。所以对于一个给定 id 为 `myBean` 的 `FactoryBean` ，调用容器的 `getBean("myBean")` 方法返回的是 `FactoryBean` 的产品；而调用 `getBean("&myBean")` 方法则返回 `FactoryBean` 实例本身。

6.10 Classpath scanning and managed components

本章中的大多数示例都使用 XML 配置元数据在 Spring 的容器中生产每一个BeanDefinition。之前的章节（6.9 节，“基于注解的容器配置”）表述了如何通过代码级的注解来提供大量的配置信息。尽管在那些示例中，“基础的”bean 的定义都是在 XML 文件中来明确定义的，而注解仅仅进行依赖注入。本节来说明另外一种通过扫描类路径的方式来隐式检测候选组件。候选组件是匹配过滤条件的类库，并有在容器中注册的对应的 bean 的定义。这就可以不用 XML 来执行 bean 的注册了，那么你就可以使用注解（比如 `@Component` ），AspectJ 风格的表达式，或者是你自定义的过滤条件来选择哪些类有在容器中注册 bean。

> 从 Spring 3.0 开始，很多由 Spring JavaConfig 项目提供的特性作为了 Spring Framework 核心的一部分。这就允许你使用 Java 而不是传统的 XML 文件来定义 bean 了。看一看 `@Configuration` ， `@Bean` ， `@Import` 和 `@DependsOn` 注解的例子来了解如何使用它们的新特性。

6.10.1 @Component and further stereotype annotations

在 Spring 2.0 版之后， `@Repository` 注解是任意满足它的角色或典型库（比如熟知的数据访问对象，DAO）的类的标记。这个标记的有多种用途，其中之一就是在 Section 19.2.2, `“Exception translation”` 中描述的异常自动转化。

Spring 2.5 引入了更多的典型注解 ： `@Component` ， `@Service` 和 `@Controller` 。 `@Component` 是对受 Spring 管理组件的通用注解。 `@Repository` ， `@Service` 和 `@Controller` 是 `@Component` 的特殊用途，比如，分别对应了持久层，服务层和表现层。因此，你可以使用 `@Component` 注解你的组件类，但是如果使用 `@Repository` ， `@Service` 或 `@Controller` 注解来替代的话，那么你的类更合适由工具来处理或与切面进行关联。比如，这些老套的注解使得理想化的目标称为切入点。而且 `@Repository` ， `@Service` 和 `@Controller` 也可以在将来 Spring Framework 的发布中携带更多的语义。因此，如果对于服务层，你在 `@Component` 或 `@Service` 中间选择的话，那么 `@Service` 无疑是更好的选择。相似地，正如上面提到的，在持久层中， `@Repository` 已经支持作为自动异常转化的标记。

6.10.2 Meta-annotations

Spring提供了很多元注解。元注解简单的说就是能被应用到另一个注解上的注解。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component // Spring will see this and treat @Service in the same way as @Component
public @interface Service {

    // ....
}
```

元注解也可以被组合使用用于创建组合注解。例如Spring MVC的 `@RestController` 注解就是 `@Controller` 和 `@ResponseBody` 。

另外，组合注解可能从元注解中任意重新声明属性来允许用户自定义。这个会特别有用当你只想暴露一个源注解的子集。例如，下面是一个自定义的 `@Scope` 注解，将作用域名称硬编码到 `@Session` 注解上，但依然允许自定义 `proxyMode` 。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Scope("session")
public @interface SessionScope {

    ScopedProxyMode proxyMode() default ScopedProxyMode.DEFAULT;
}
```

`@SessionScope` 可以不声明 `proxyMode` 就使用，如下所示：

```
@Service
@SessionScope
public class SessionScopedUserService implements UserService {
    // ...
}
```

或者为 `proxyMode` 重载一个值，如下所示：

```
@Service
@SessionScope(proxyMode = ScopedProxyMode.TARGET_CLASS)
public class SessionScopedService {
    // ...
}
```

更多详情，请参阅 `37. Spring Annotation Programming Model`

6.10.3 Automatically detecting classes and registering bean definitions

Spring可以自动检测固有的类并在`ApplicationContext`中注册 对应的`BeanDefinition`。比如，下面的两个类就是自动检测的例子：

```
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

}
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

要自动检测这些类并注册对应的bean，你需要添加`@ComponentScan`到你的`@Configuration`类上，其中的`base-package`元素是这两个类的公共父类包。（你可以任意选择使用逗号/分号/空格分隔的列表来将每个类引入到父包。）

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    ...
}
```

> 为了更简洁，上面的示例可以使用注解的`value`属性，也就是`ComponentScan("org.example")`

下面的示例使用XML配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

> `<context:component-scan>` 隐式地开启了 `<context:annotation-config>` 功能，当使用了 `<context:component-scan>` 时通常没必要再包含 `<context:annotation-config>`。
>
> 对类路径包的扫描需要类路径下存在对应的目录实体。当你使用 Ant 来构建 JAR 包时，要保证你没有激活 JAR 目标中的 files-only 开关。同时，在一些环境中基于安全策略，类路径下的文件不会被暴露出来，如果单独的apps在JDK 1.7.0_45和更高的版本（这需要'Trusted-Library'计划在你的manifests中，see http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources）

此外，当你使用 `component-scan` 时， `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 二者是隐式包含着的。这就意味着两个组件被自动检测之后就装配在一起了-而不需要在 XML 中提供其它任何 bean 的配置元数据。

> 你可以将 annotation-config 属性置为 false 来关闭 `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 注册。

6.10.4 Using filters to customize scanning

默认情况下，使用 `@Component` ， `@Repository` ， `@Service` ， `@Controller` 注解或使用了进行自定义的 `@Component` 注解的类本身仅仅检测候选组件。你可以修改并扩展这种行为，仅仅应用自定义的过滤器就可以了。在 `@ComponentScan` 注解中添加 `include-filter` 或 `exclude-filter` 参数就可以了（或者作为component-scan元素的include-filter 或 exclude-filter 子元素）。每个过滤器元素需要 `type` 和 `expression` 属性。下面的表格描述了过滤选项。

表 **6.5**过滤器类型

过滤器类型	表达式示例	描述
annotation（注解默认）	org.example.SomeAnnotation	使用在目标组件的类级别上
assignable（分配）	org.example.SomeClass	目标组件分配去（扩展/实现）的类（
aspectj	org.example..*Service+	AspectJ 类型表达式来匹配目标组件
regex（正则表达式）	org.example.Default.*	正则表达式来匹配目标组件类的名称
custom（自定义）	org.example.MyTypeFilter	自定义 `org.springframework.core.type.Type` 接口的实现类

下面的示例代码展示了 XML 配置忽略所有 `@Repository` 注解并使用“sub”库来替代。

```
@Configuration
@ComponentScan(basePackages = "org.example",
        includeFilters = @Filter(type = FilterType.REGEX, pattern = ".*Stub.*Repositor
y"),
        excludeFilters = @Filter(Repository.class))
public class AppConfig {
    ...
}
```

一样的可以使用XML配置：

```
<beans>
    <context:component-scan base-package="org.example">
        <context:include-filter type="regex"
                expression=".*Stub.*Repository"/>
        <context:exclude-filter type="annotation"
                expression="org.springframework.stereotype.Repository"/>
    </context:component-scan>
</beans>
```

> 你可以使用注解的 `useDefaultFilters=false` 或 `<component-scan/>` 元素中的 `use-default-filter="false"` 属性来关闭默认的过滤 器。这会关闭自动检测 `@Component` ，`@Repository` ，`@Service` 或 `@Controller` 注解的类。

6.10.5 Defining bean metadata within components

Spring 组件可以为容器提供 bean 定义的元数据。你可以在 `@Configuration` 注解的类中使用 `@Bean` 注解来达成这一目的。这里有一个简单的示例：

```
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }

}
```

这个类是个 Spring 组件，它的 `doWork()` 方法中包含特定应用代码。同时，它也提供了 bean 的定义并且由工厂方法来指向 `publicInstance()` 方法。 `@Bean` 注解定义了工厂 方法和其它 bean 定义的属性，比如通过 `@Qualifier` 注解表示的限定符。其它方法级的注解可以使用的是 `@Scope` ， `@Lazy` 和自定义限定符注解。

> 除了组件初始化的角色， `@Lazy` 注解也可以跟着 `@Autowired` 或 `@Inject` 放置到注入点。在这种上下文中，它将生成一个延迟解析的代理注入。

自动装配字段和方法也是支持的，这在之前讨论过，而且还有对自动装配 `@Bean` 方法的支持：

```
@Component
public class FactoryMethodComponent {

    private static int i;

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    // use of a custom qualifier and autowiring of method parameters

    @Bean
    protected TestBean protectedInstance(
            @Qualifier("public") TestBean spouse,
            @Value("#{privateInstance.age}") String country) {
        TestBean tb = new TestBean("protectedInstance", 1);
        tb.setSpouse(spouse);
        tb.setCountry(country);
        return tb;
    }

    @Bean
    @Scope(BeanDefinition.SCOPE_SINGLETON)
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TA
RGET_CLASS)
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }

}
```

这个示例为另外一个名为 `privateInstance` 的bean的 `Age` 属性自动装配了 `String` 方法参数 `country` 。Spring 的表达式语言元素通过 `#{<expression>}` 表示定义了属性的值。对于 `@Value` 注解，当解析表达式文本时，表达式解析器会预先配置来查看 bean 的名称。Spring组件中的 `@Bean` 方法会被不同方式处理，而不会像Spring的 `@Configuration` 类中的同仁那样。不同的是 `@Component` 类没有使用 CGLIB 来加强并拦截字段和方法的调用。CGLIB 代理是调用 `@Configuration` 类中的 `@Bean` 方法或字段来创建 bean 元数据引用协作对象的手段。方法没有使用通常的 Java 语义来调用。相比之下，调用普通的 `@Component` 类中的 `@Bean` 方法或字段有标准的 Java 语义，没有特殊的CGLIB处理或其他的限制应用。

> 你可能声明 `@Bean` 为 `static` ，允许包含它们的配置类没有创建为实例时进行调用。这使得定义后置处理器特别有意义，例如， `BeanFactoryPostProcessor` 或 `BeanPostProcessor` ，由于这样的bean将在容器生命周期早期初始化，因此应该避免在该点触发配置的其他部分。
> 注意，调用静态的 `@Bean` 方法将不会被容器拦截，即使是在 `@Configuration` 类里（看上面）。这是由于技术上的局限性：CGLIB 的子类仅仅可以重载非静态的方法。因此，直接调用另一个 `@Bean` 方法将会有标准的Java语义，从而直接从工厂方法返回一个独立的实例。
> 在spring容器中， `@Bean` 方法的Java语言可视性并没有直接对bean的定义产生影响。你可以自由地声明你自己的工厂方法填充到非 `@Configuration` 类中，也可以是静态方法。当然，在 `@Configuration` 类中合格的 `@Bean` 方法应该是可重写的，也就是说，你不应该声明为 `private` 或 `final` 类型。 最后， `@Bean` 也可以用在给定组件或配置类的基类上，以及Java 8的默认方法声明的被组件或配置类实现的接口。这使得更灵活地组成复杂的配置结构，甚至在Spring 4.2，多重继承可以通过java 8的默认方法实现。
>
> ## 6.10.6 Naming autodetected components
>
> 当组件被自动检测作为扫描进程的一部分时，它的bean名称是由 `BeanNameGenerator` 策略来生成并告知扫描器的。默认情况下，Spring 构造型的注解（ `@Component` ， `@Repository` ， `@Service` 和 `@Controller` ）包含 `name` 值将会提供该值给对应 bean 定义的名称。 如果注解不包含 `name` 值或任何被检测到的组件（比如那些被自定义过滤器发现的），默认的 bean 的名称生成器返回未大写的非限定符类名。比如，如果下面的两个组件被检测到了，那么名称可能是 myMovieLister 和 movieFinderImpl：
>
> ```
> @Service("myMovieLister")
> public class SimpleMovieLister {
>     // ...
> }
> @Repository
> public class MovieFinderImpl implements MovieFinder {
>     // ...
> }
> ```

如果你不想使用默认的 bean 命名策略，你可以提供自定义的 bean 命名策略。首先，实现 `BeanNameGenerator` 接口，要保证包含默认的无参构造器。之后，在配置扫描器时，要提供类的完全限定名。

```
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.clas
s)
public class AppConfig {
    ...
}
```

```
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

作为通用的规则，要考虑使用注解指定名称时，其它组件可能会有对它的明确的引用。另一方面，当容器负责装配时，自动生成名称是可行的。

6.10.7 Providing a scope for autodetected components

一般情况下，Spring 管理的组件，自动检测组件默认和最多使用的作用域是单例。然而，有时你需要其它作用域，Spring 2.5 提供了一个新的 `@Scope` 注解。仅仅需要提供作用域的名称到注解上：

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

为作用域提供自定义策略而不是基于注解的方式，实现 `ScopeMetadataResolver` 接口，并保证包含了默认的无参构造方法。之后，当配置扫描器时，要提供类的完全限定名：

```
@Configuration
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.clas
s)
public class AppConfig {
    ...
}
<beans>
    <context:component-scan base-package="org.example"
            scope-resolver="org.example.MyScopeResolver" />
</beans>
```

> 当使用确定的非单例作用域时，它可能必须要为该作用域的对象生成代理。这个原因在“Scoped beans as dependencies”章节中描述过了。出于这样的目的，在`component-scan`元素中可以使用`scoped-proxy`属性。三种可能的值是：`no（无）`，`interface（接口）`和`targetClass（目标类）`。比如，下面的配置就会启动标准的 JDK 动态代理：
>
> ```
> @Configuration
> @ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERF
> ACES)
> public class AppConfig {
>     ...
> }
> <beans>
>     <context:component-scan base-package="org.example"
>         scoped-proxy="interfaces" />
> </beans>
> ```
>
> ## 6.10.8 Providing qualifier metadata with annotations
>
> `@Qualifier`注解在 “Fine-tuning annotation-based autowiring with qualifiers”章节中讨论过了。那章节中演示了`@Qualifier`注解的使用和当你需要处理自动装配候选者时，自定义限定符注解来提供细粒度控制。因为那些示例是基于 XML 的 bean 定义的，限定符元数据在候选者 bean 定义中提供，并使用了 XML 中的 bean 元素的`qualifier`和`meta`子元素。 当对自动检测组件使用基于类路径扫描时，你可以在候选者类中使用类型级别的注解提供限定符元数据。下面的三个示例就展示了这个技术：
>
> ```
> @Component
> @Qualifier("Action")
> public class ActionMovieCatalog implements MovieCatalog {
>     // ...
> }
> @Component
> @Genre("Action")
> public class ActionMovieCatalog implements MovieCatalog {
>     // ...
> }
> @Component
> @Offline
> public class CachingMovieCatalog implements MovieCatalog {
>     // ...
> }
> ```
>
> 对于大多数基于注解的方式，要记得注解元数据会绑定到类定义本身中去，而使用 XML 就允许对多个相同类型的 bean 在它们的限定符元数据中提供变化，因为那些元数据是对于每个实例而不是每个类提供的。

6.11 Using JSR 330 Standard Annotations

从 Spring 3.0 开始，Spring 提供了对 JSR-330 标准注解（依赖注入）的支持。这些注解可以和 Spring 注解以相同方式被扫描到。你仅仅需要在类路径下添加相关的 jar 包即可。

如果你使用Maven，那么标准Maven仓库[http://repo1.maven.org/maven2/javax/inject/javax.inject/1/] 中 `javax.inject` 的artifact是可用的。你仅仅需要在 pom.xml 中添加如下的依赖即可：

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

6.11.1 Dependency Injection with @Inject and @Named

取代 `@Autowired` ， `javax.inject.Inject` 还可以像下面这样使用：

```
import javax.inject.Inject;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...

}
```

与 `@Autowired` 一样，可以在类级别，字段级别，方法级别和构造方法参数级别使用 `@Inject` 。如果你想对被注入的依赖使用限定符名称，你应该按如下方式使用 `@Named` 注解：

```
import javax.inject.Inject;
import javax.inject.Named;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...

}
```

6.11.2 @Named: a standard equivalent to the @Component annotation

取代 `@Component` ， `javax.inject.Named` 还可以像下面这样使用：

```
import javax.inject.Inject;
import javax.inject.Named;

@Named("movieListener")
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...

}
```

使用 `@Component` 而不指定组件的名称是很常用的方式。 `@Named` 可以被用于相同的 情况：

```
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...

}
```

当使用 `@Named` 时，可以使用相同的方式使用Spring注解的组件扫描：

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    ...
}
```

6.11.3 Limitations of the standard approach

当使用标准注解时，了解一些很重要的特性是不可用的是很有必要的，在下面的表格中给出： 表 6.6 Spring注解和标准注解的对比

Spring	javax.inject.*	javax.inject 限制/ 注释
@Autowired	@Inject	@Inject 没有 'required' 属性
@Component	@Named	
@Scope(" singleton ")	@Singleton	jsr-330 默认范围和 Spring 的 prototype相似。但是，要保持和 Spring 一般默认值一致，在 Spring 容器中 jsr-330 的 bean 声明默认是 singleton 的。要使用另外的范围，你应该使用 Spring 的@Scope 注解。javax.inject 也供@Scope 注解。不过这仅仅用于创建你自己的注解。
@Qualifier	@Named	
@Value	-	不等同
@Required	-	不等同
@Lazy	-	不等同

6.12 Java-based container configuration

6.12.1 Basic concepts: @Bean and @Configuration

Spring 中新的 Java 配置支持的核心就是 `@Configuration` 注解的类和 `@Bean` 注解的方法。

`@Bean` 注解用来指定一个方法实例，配置和初始化一个新对象交给Spring IOC容器管理。对于那些熟悉Spring `<beans>` XML配置的人来说，`@Bean` 注解和 `<bean>` 元素扮演相同的角色。你可以使用在 `@Component` 类中使用 `@Bean` 注解方法，但更常用的，是在 `@Configuration` 类中使用。

`@Configuration` 注解的类表示它的主要目的是作为bean定义的来源。另外，`@Configuration` 类允许内部bean依赖通过简单地调用同一类内的其他 `@Bean` 方法进行定义。最简单的 `@Configuration` 类可能是这样的：

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

}
```

上面 `AppConfig` 类与下面Spring XML中的配置是等同的：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

> Full @Configuration vs 'lite' @Beans mode? 当 `@Bean` 方法不是在 `@Configuration` 中声明时，被称为‘lite’ mode。例如，bean方法在 `@Component` 或甚至在简单的PO类中声明，都被认为是’lite‘。
> 与full `@Configuration` 不同的是，lite `@Bean` 方法无法声明内部bean依赖。通常，当在 `lite` mode下时一个 `@Bean` 方法不应该调用另一个 `@Bean` 方法。
> 只有在 `@Configuration` 类下使用 `@Bean` 方法是被推荐的，确保'full'mode经常使用到。这将阻止被同一个 `@Bean` 方法意外地多次调用，帮助减少在 `lite` mode下出现的难以追踪的bugs。

`@Bean` 和 `@Configuration` 注解将会在下面的章节中被深入地探讨。首先，我们先来看看使用基于 Java 的配置创建 Spring 容器的各种方 式。

6.12.2 Instantiating the Spring container using AnnotationConfigApplicationContext

下面的章节讲解了 Spring 的 `AnnotationConfigApplicationContext` ，是在 Spring3.0 中新加入的。这个全能的 `ApplicationContext` 实现类不仅仅可以接受 `@Configuration` 类作为输入，也可以是普通的 `@Component` 类，还有使用 JSR-330 元数据注解的类。 当 `@Configuration` 类作为输入时， `@Configuration` 类本身作为 bean 被注册了，并且类内所有声明的 `@Bean` 方法也被作为 bean 注册了。 当 `@Component` 和 JSR-330 类作为输入时，它们被注册为 bean，并且被假设如 `@Autowired` 或 `@Inject` 的 DI 元数据在类中需要的地方使用。

简单构造

与使用Spring XML配置作为输入实例化 `ClassPathXmlApplicationContext` 过程类似，当实例化 `AnnotationConfigApplicationContext` 时 `@Configuration` 类可能作为输入。这就允许在 Spring 容器中完全可以不使用 XML：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

正如上面所提到的， `AnnotationConfigApplicationContext` 不仅仅局限于和 `@Configuration` 类合作。任意 `@Component` 或 JSR-330 注解的类都可以作为构造方法的输入。比如：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl.clas
s, Dependency1.class, Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

上面假设 `MyServiceImpl` ， `Dependency1` 和 `Dependency2` 使用了 Spring 依赖注入注解，比如 `@Autowired` 。

使用 **register(Class<?>...)**编程式构建容器

`AnnotationConfigApplicationContext` 可以使用无参构造方法来实例化，之后使用 `register()` 方法来配置。这个方法当编程式地构建 `AnnotationConfigApplicationContext` 时尤其有用。

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

使用 scan(String..)开启组件扫描

要开启组件扫描，仅仅需要在你的 `@Configuration` 类上加上如下注解：

```
@Configuration
@ComponentScan(basePackages = "com.acme")
public class AppConfig  {
    ...
}
```

有经验的 Spring 用户肯定会熟悉下面这个 Spring 的 `context:` 命名空间中的常用 XML声明：

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

在上面的示例中， `com.acme` 包会被扫描到，去查找任意 `@Component` 注解的类，那些类就会被注册为 Spring 容器中的 bean。 `AnnotationConfigApplicationContext` 暴露出 `scan(String ...)` 方法，允许相同的组件扫描功能：

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

> 记得 `@Configuration` 类是使用 `@Component` 进行元数据注解的，所以它们是组件扫描的候选者！在上面的示例中，假设 `AppConfig` 是声明在 `com.acme` 包（或是其中的子包）中的，那么会在调用 `scan()` 方法时被找到，在调用 `refresh()` 方法时，所有它的 `@Bean` 方法就会被处理并注册为容器中的 bean。

支持Web应用的 `AnnotationConfigWebApplicationContext`

`WebApplicationContext` 是 `AnnotationConfigApplicationContext` 的变种，适用于 `AnnotationConfigWebApplicationContext`。当配置 Spring 的 Servlet 监听器 `ContextLoaderListener`，Spring MVC 的 `DispatcherServlet` 等时，这个实现类就可能被用到了。下面的代码是在 `web.xml` 中的片段，配置了典型的 Spring MVC 的 Web 应用程序。注意 `contextClass` 上下文参数和初始化参数的使用：

```
<web-app>
    <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.web.context.support.AnnotationConfigWebApplicationCont
ext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-delimited
        fully-qualified @Configuration classes. Fully-qualified packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using ContextLoaderListener -
->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listene
r-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-clas
s>
        <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext
            instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.web.context.support.AnnotationConfigWebApplication
Context
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more comma- or space-delim
ited
            and fully-qualified @Configuration classes -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
```

```
            <param-value>com.acme.web.MvcConfig</param-value>
        </init-param>
    </servlet>

    <!-- map all requests for /app/* to the dispatcher servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>
</web-app>
```

6.12.3 Using the @Bean annotation

`@Bean` 是一个方法级的注解，与XML的 `<bean/>` 元素功能相同。该注解支持一些 `<bean/>` 上的属性，如: `init-method` , `destroy-method` , `autowiring` 和 `name` 。你可以在 `@Configuration` 或 `@Component` 类里使用 `@Bean` 注解。

声明一个bean

要声明一个bean，可以简单地使用 `@Bean` 注解到一个方法上。你可以在 `ApplicationContext` 容器中使用这个方法的返回值来注册一个bean定义。默认的，bean的名称将会与方法的名称相同。下面是一个 `@Bean` 声明的简单示例：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }

}
```

上述配置是完全与下面的Spring XML相等的：

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

两者的声明都在 `ApplicationContext` 中创建了一个名为 `transferService` 的bean，绑定了一个 `TransferServiceImpl` 类型的实例对象：

> transferService -> com.acme.TransferServiceImpl

Bean依赖

`@Bean` 注解的方法可以有任意数量的参数依赖来构建bean。比如，如果我们的 `TransferService` 要求一个 `AccountRepository` ，我们可以具体化该依赖通过一个方法参数：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }

}
```

这个解析机制与基于构造器的依赖注入非常相似，详情查看“Constructor-based dependency injection”章节。

接收生命周期回调

任何使用 `@Bean` 注解定义的类都支持定期的生命周期回调，可以使用 JSR-250 的 `@PostConstruct` 和 `@PreDestroy` 注解，详情查看 `6.9.8 @PostConstruct and @PreDestroy`

定期的生命周期回调得到了很好地支持。如果一个bean实现了 `InitializingBean` , `DisposableBean` , or `Lifecycle` 接口，他们各自的方法将会被容器调用。

标准的一套 `*Aware` 接口，例如， `BeanFactoryAware` , `BeanNameAware` , `MessageSourceAware` , `ApplicationContextAware` 等等，也一样得到了很好的支持。

`@Bean` 注解支持指定任意的初始化和销毁的回调方法，非常类似于Spring XML中bean元素的 `init-method` 和 `destroy-method` 属性：

```
public class Foo {
    public void init() {
        // initialization logic
    }
}

public class Bar {
    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public Foo foo() {
        return new Foo();
    }

    @Bean(destroyMethod = "cleanup")
    public Bar bar() {
        return new Bar();
    }
}
```

默认地，使用Java配置的bean定义有public的 `close` 或 `shutdown` 方法时，在销毁回调时会被自动地调用。所以如果你有public的 `close` 或 `shutdown` 方法，而你不想它在容器关闭时被调用，那你可以简单地加上 `@Bean(destroyMethod="")` 到你的bean定义上，以此来使默认的 `(inferred)` 行为失效。

你可以想默认地通过JNDI获取你需要的在应用之外管理的资源作为它的生命周期。特别地，一定要始终做一个 `DataSource` ，因为它是已知的有问题的：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

当然，上面 `Foo` 的示例，它等同于在构造期间调用 `init()` 方法：

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.init();
        return foo;
    }

    // ...
}
```

> 当你完全使用Java配置，你对你的对象可以做任何你想做的，不要总必须依赖于容器的生命周期！

指定bean的作用域

使用 `@Scope` 注解

你可以指定使用 `@Bean` 注解的bean定义应该有个指定的作用域。你可以使用任何在6.5章节Bean scopes中提到的标准的作用域。

默认的作用域是 `singleton` ，但你可以使用 `@Scope` 注解进行覆盖：

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }

}
```

@Scope 和 scoped-proxy

Spring 提供了一个便利的途径，通过作用域代理处理作用域依赖。当使用XML配置时，最简单的方法创建这样一个代理就是使用 `<aop:scoped-proxy/>` 元素。在Java中使用 `@Scope` 注解带 `proxyMode` 属性来配置你的beans是等价的。默认的是没有代理（ `ScopedProxyMode.NO` ），但你可以指定 `ScopedProxyMode.TARGET_CLASS` 或 `ScopedProxyMode.INTERFACES` 。

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public UserPreferences userPreferences() {
    return new UserPreferences();
}

@Bean
public Service userService() {
    UserService service = new SimpleUserService();
    // a reference to the proxied userPreferences bean
    service.setUserPreferences(userPreferences());
    return service;
}
```

自定义**bean**命名

默认地，配置类使用 `@Bean` 方法的名称来作为注册bean的名称。这个方法可以被重写，当然，使用的是 `name` 属性。

```
@Configuration
public class AppConfig {

    @Bean(name = "myFoo")
    public Foo foo() {
        return new Foo();
    }

}
```

Bean 混淆现象

在 6.3.1 , “Naming beans” 章节讨论过，有时需要对单个bean使用多个名称才能满足需要，否则就会有bean混淆现象。 `@Bean` 注解的 `name` 属性为了达到该目的可以接收一个字符串数组。

```
@Configuration
public class AppConfig {

    @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }

}
```

Bean 描述

有时提供一个更详细的bean的文本描述是很有帮助的。这是特别有用当beans用于（可能通过JMX）监测的目的。

使用 `@Description` 注解来对 `@Bean` 添加一个描述：

```
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Foo foo() {
        return new Foo();
    }

}
```

6.12.4 Using the @Configuration annotation

`@Configuration` 是一个类级别的注解，用于表明此对象是一个bean定义的资源。 `@Configuration` 类通过public的 `@Bean` 注解的方法来声明beans。调用 `@Configuration` 类的 `@Bean` 方法也可以被用于定义inter-bean依赖。详情查看 `6.12.1, “Basic concepts: @Bean and @Configuration”章节` 。

Injecting inter-bean dependencies

当 `@Beans` 上有其它bean的依赖，简单演示一个bean方法调用另一个：

```
@Configuration
public class AppConfig {

    @Bean
    public Foo foo() {
        return new Foo(bar());
    }

    @Bean
    public Bar bar() {
        return new Bar();
    }

}
```

在上面的例子中， `foo` bean通过构造器注入接收了一个 `bar` 引用。

> 这个声明inter-bean依赖的方法只会在`@Bean`声明在`@Configuration`类下时才会起效。你不能使用简单的`@Component`类声明inter-bean依赖。

查找方法注入

如前所述，查找方法注入是一个高级特性，你应该很少会用到。当一个singleton作用域的bean有一个prototype作用域bean的依赖时，这将很有用。对于Java配置，提供了一种自然的方式实现这一模式：

```
public abstract class CommandManager {
    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();

        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
    return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

使用Java配置支持，你可以创建一个`CommandManager`的子类，其虚方法`createCommand()`将会被重写，通过此种方式，来查找一个新的（prototype）的command对象。

```
@Bean
@Scope("prototype")
public AsyncCommand asyncCommand() {
    AsyncCommand command = new AsyncCommand();
    // inject dependencies here as required
    return command;
}

@Bean
public CommandManager commandManager() {
    // return new anonymous implementation of CommandManager with command() overridden
    // to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
        }
    }
}
```

Further information about how Java-based configuration works internally

下面的例子展示了 `@Bean` 注解的方法被调用了2次：

```
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }

}
```

`clientDao()` 在 `clientService1()` 中被调用一次，在 `clientService2()` 也被调用一次。从这个方法创建一个 `ClientDaoImpl` 实例并返回它，你通常期望会生成2个实例（每个service一个）。那肯定是有问题的：在Spring中，实例化beans默认拥有一个singleton作用域。这就是神奇的地方：所有的 `@Configuration` 类在启动的阶段使用CGLIB进行子类化。在子类中，子方法首先检查容器中任何缓存（scoped）的beans在它调用父方法及创建新的实例之前。注意，Spring3.2已经不再需要添加CGLIB到你的classpath中了，因为CGLIB类已经被打包到 `org.springframework` ，并被包含到 `spring-core` JAR包中了。

> 根据你的bean的作用域的不同，行为可能会有所不同。我们这里仅谈论了singltons。
>
> 由于CGLIB在启动期间动态添加特性，所以这里有几个限制：
>
> - 配置类必须不能为final
> - 应该有一个无参的构造器

6.12.5 Composing Java-based configurations

使用 **`@Import`** 注解

就像 Spring 的 XML 文件中使用的 `<import/>` 元素帮助模块化配置一样，`@Import` 注解允许从其它配置类中加载 `@Bean` 的配置：

```
@Configuration
public class ConfigA {

     @Bean
    public A a() {
        return new A();
    }

}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }

}
```

现在，当实例化上下文时，不需要指定 `ConfigA.class` 和 `ConfigB.class` 了，仅仅 `ConfigB` 需要被显式提供：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

    // now both beans A and B will be available...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

这种方式简化了容器的实例化，仅仅是一个类需要被处理，而不是需要开发人员在构造时记住很多大量的 `@Configuration` 类。

在引入的**@Bean**定义中注入依赖

上面的示例是可行的，但是太过简单。在很多实际场景中，bean 会有依赖其它配置的类的依赖。当使用 XML 时，这本身不是什么问题，因为没有调用编译器，而且我们可以仅仅声明 `ref="someBean"` 并且相信 Spring 在容器初始化时可以完成。当然，当使用 `@Configuration` 的类时，Java 编译器在配置模型上放置约束，对其它 bean 的引用必须是符合 Java 语法的。

幸运的是，解决这个问题非常简单。我们已经讨论过了，`@Bean` 方法可以有任意个参数声明bean依赖。让我们考虑一个更真实的场景，有个 `@Configuration` 类，每个bean都依赖其他的配置中声明的bean：

```
@Configuration
public class ServiceConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }

}

@Configuration
public class RepositoryConfig {

    @Bean
    public AccountRepository accountRepository(DataSource dataSource) {
        return new JdbcAccountRepository(dataSource);
    }

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.c
lass);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

有另外一种方法可以达到一样的结果。记住 `@Configuration` 类最终是容器中的另外一个bean：也就是说它们可以像其它 bean 那样利用 `@Autowired` 注入元数据！

> 要确定这种方式的依赖注入是最简单的那种。`@Configuration` 类在容器初始化时就进行处理了，强制要求依赖使用这种方式进行注入可能导致意外的早期初始化问题。如果可能，就采用如上述例子所示的基于参数的注入。 同时，也要特别小心通过 `@Bean` 的 `BeanPostProcessor` `BeanFactoryPostProcessor` 定义。它们应该被声明为static的 `@Bean` 方法，不会触发实例化它们的包含类。否则，`@Autowired` 和 `@Value` 将在配置类上不生效，因为它太早被创建为一个实例了。

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }

}

@Configuration
public class RepositoryConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.c
lass);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

在上面的场景中，使用 `@Autowired` 工作正常，提供所需的模块化，但是准确地决定在哪儿声明自动装配的 bean 还是有些含糊。比如，作为开发者来看待 `ServiceConfig` ，你如何准确知道 `@Autowired AccountRepository` 在哪里声明的？它没有显式地出现在代码中，这可能很不

错。要记得 `SpringSource Tool Suite` 提供工具可以生成展示所有对象是如何装配起来的-那可能就是你所需要的。而且，你的 Java IDE 可以很容器发现所有的声明，还有使用的 `AccountRepository` 类型，也会很快地给你展示出`@Bean`方法的位置和返回的类型。

在这种歧义不能接受，你想直接在 IDE 中从一个`@Configuration` 类导航到另一个，要考虑自动装配配置类的本身：

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        // navigate 'through' the config class to the @Bean method!
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }

}
```

在上面的情形中，定义`AccountRepository`是完全明确的。而`ServiceConfig`却紧紧耦合到`RepositoryConfig`中了；这就需要我们来权衡了。这种紧耦合可以使用基于接口或抽象基类的`@Configuration`类来减轻。考虑下面的代码：

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}

@Configuration
public interface RepositoryConfig {

    @Bean
    AccountRepository accountRepository();

}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig {

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }

}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) // import the concrete c
onfig!
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.c
lass);
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

现在 `ServiceConfig` 和 `DefaultRepositoryConfig` 的耦合就比较松了，并且内建的 IDE 工具也一直有效：对于开发人员来说会更加简单地获取 `RepositoryConfig` 实现类的类型层次。以这种方式，导航 `@Configuration` 类和它们的依赖就和普通的基于接口代码的导航过程没有任何区别了。

Conditionally include @Configuration classes or @Bean methods

基于任意的系统状态，使完整的 `@Configuration` 类或单独的 `@Bean` 方法有条件性地生效或失效是很有用的。一个常见的例子就是当一个指定的profile已经在Spring环境中生效时使用 `@Profile` 注解来激活beans（详见 `Section 6.13.1, “Bean definition profiles”`）。

`@Profile` 注解是通过使用一个非常灵活的称为 `@Conditional` 的注解实现的。`@Conditional` 注解表示具体的 `org.springframework.context.annotation.Condition` 实现者应该在 `@Bean` 被注册前先被访问。

`Condition` 接口的实现简单地提供了一个 `matches(…)` 方法，用来返回 `true` 或 `false`。例如，以下就是使用 `@Profile` 具体的 `Condition` 实现：

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
        // Read the @Profile annotation attributes
        MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Prof
ile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(((String[]) value))) {
                    return true;
                }
            }
            return false;
        }
    }
    return true;
}
```

结合 Java和 和 XML

Spring 的 `@Configuration` 类并不是完全100%地支持 Spring XML 替换的。一些基本特性，比如 Spring XML 的命名空间会保持在一个理想的方式下去配置容器。在 XML 更便于使用或是必须要使用的情况下，你也有另外一个选择：以“XML为中心”的方式来实例化容器，比如，`ClassPathXmlApplicationContext`，或者以“Java 为中心”的方式，使用 `AnnotationConfigurationApplicationContext` 和 `@ImportResource` 注解来引入需要的 XML。

以“XML为中心”使用@Configuration

以XML配置文件结合包含 `@Configuration` 类的点对点的方式来启动 Spring 容器是更好的选择。比如，在使用了 Spring XML配置的大型的代码库中，根据需要从已有的 XML 文件中创建 `@Configuration` 类是很简单的。在下面，你会发现在这种“XML为中心”情形下，使用 `@Configuration` 类的选项。 记住， `@Configuration` 类最终仅仅是容器中的bean。在这个示例中，我们创建了名为 `AppConfig` 的 `@Configuration` 类，并且将它包含在 `system-test-config.xml` 文件中作为 `<bean/>` 的定义。因为开启了 `<context:annotation-config/>` 配置，容器会识别 `@Configuration` 注解，并以合适的方式处理声明在 `AppConfig` 中 `@Bean` 方法。

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }

}
```

system-test-config.xml:

```
<beans>
    <!-- enable processing of annotations such as @Autowired and @Configuration -->
    <context:annotation-config/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="com.acme.AppConfig"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

> jdbc.properties:
> jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
> jdbc.username=sa
> jdbc.password=

```
public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:/com/acme/s
ystem-test-config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

> 在上面的 `system-test-config.xml` 文件中， `AppConfig` 的 `<bean/>` 定义没有声明 `id` 元素。这么做也是可以接受的，就不必让其它 bean 去引用它了，同时也就不可能从容器中通过明确的名称来获取它了。同样地， `DataSource` bean，通过类型自动装配，那么明确的 bean `id` 就不严格要求了。

因为 `@Configuration` 是使用 `@Component` 来元数据注解的，被 `@Configuration` 注解的类是自动作为组件扫描的候选者的。使用上面相同的场景，我们可以重新来定义 `system-test-config.xml` 文件来利用组件扫描的优点。注意这种情况下，我们不需要明确地声明 `<context:annotation-config/>` ，因为 `<context:component-scan/>` 开启了相同的功能。

system-test-config.xml:

```
<beans>
    <!-- picks up and registers AppConfig as a bean definition -->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

使用了**@ImportResource**导入**XML**的**@Configuration**类为中心

在 `@Configuration` 类作为配置容器主要机制的应用程序中，使用一些 XML 还是必要的。在这些情况中，仅仅使用 `@ImportResource` 来定义 XML 就可以了。这么来做就实现了“Java为中心”的方式来配置容器并保持 XML 在最低限度。

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }

}
```

```
properties-config.xml
<beans>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>
</beans>
```

> jdbc.properties
> jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
> jdbc.username=sa
> jdbc.password=
>
> ```
> public static void main(String[] args) {
>     ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.cla
> ss);
>     TransferService transferService = ctx.getBean(TransferService.class);
>     // ...
> }
> ```

环境的抽象

`Environment` 是一个集成到容器之中的特殊抽象，它针对应用的环境建立了两个关键的概念：`profile` 和 `properties` .

*profile*是命名好的，其中包含了多个Bean的定义的一个逻辑集合，只有当指定的profile被激活的时候，其中的Bean才会激活。无论是通过XML定义的还是通过注解解析的Bean都可以配置到profile之中。而 `Environment` 对象的角色就是跟profile相关联，然后决定来激活哪一个profile，还有哪一个profile为默认的profile。

*properties*在几乎所有的应用当中都有着重要的作用，当然也可能存在多个数据源：property文件，JVM系统property，系统环境变量，JNDI，servlet上下文参数，ad-hoc属性对象，Map等。 `Environment` 对象和property相关联，然后来给开发者一个方便的服务接口来配置这些数据源，并正确解析。

Bean定义的profile

在容器之中，Bean定义profile是一种允许不同环境注册不同bean的机制。环境的概念就意味着不同的Bean对应不同的开发者，而且这个特性在以下场景使用十分便利：

- 解决一些内存中的数据源的问题，可以在不同环境访问不同的数据源，开发环境，QA测试环境，生产环境等。
- 仅仅在开发环境来使用一些监视服务
- 在不同的环境，使用不同的bean实现

下面参考一个例子，下面的应用需要一个 `DataSource` ，在一个测试的环境下，可能类似如下代码：

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

现在考虑如果应用部署到QA环境或者生产环境，假设应用的数据源是服务器上的JNDI目录的话，我们的 `DataSource` 可能会如下：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
}
```

问题就是如何基于当前的环境来使用不同的配置。在以前，Spring的开发者开发了很多的方法来解决这个问题，通常都依赖于系统环境变量和XML中的 `<import/>` 标签以及占位符 `${placeholder}` 等来根据不同的环境解析当前的配置文件。现在Bean的profile属于容器的特性，也是该问题的解决方案之一。

如果我们泛化了我们一些特殊环境下引用的bean定义，我们可以将其中指定的Bean注入到特定的context之中，而不是所有的context之中了。很多开发者就希望能够在一种环境下使用Bean定义A，另一种情况下使用Bean定义B。

`@Profile` 注解

`@Profile` 注解允许开发者来表示一个组件是否适合在当前环境来进行注册，只有当前的Profile是激活的时候，对应的Bean才会被注册到上下文中。使用前面的例子，代码可以进行如下调整:

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

`@Profile` 注解可以当做元注解来使用。比如，下面所定义的 `@Production` 注解就可以来替代 `@Profile("production")`：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

`@Profile` 注解也可以在方法级别使用，可以声明在包含 `@Bean` 注解的方法之上：

```
@Configuration
public class AppConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }

    @Bean
    @Profile("production")
    public DataSource productionDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

> 如果配置了 `@Configuration` 的类同时配置了 `@Profile` ，那么所有的配置了 `@Bean` 注解的方法和 `@Import` 注解的相关的类都会被传递为该 `Profile` 。除非这个 `Profile` 激活了，否则其中的Bean定义都不会激活。如果配置为 `@Component` 或者 `@Configuration` 的类标记了 `@Profile({"p1", "p2"})`,那么这个类当且仅当 `Profile` 是p1或者p2的时候才会激活。如果某个 `Profile` 的前缀是 `!` 这个非操作符，那么 `@Profile` 注解的类会只有当前的Profile没有激活的时候才能生效。举例来说，如果配置为 `@Profile({"p1", "!p2"})` ，那么注册的行为会在Profile为 `p1` 或者是Profile为非 `p2` 的时候才会激活。

XML中Bean定义的profile

在XML中相对应配置是 `<beans/>` 中的 `profile` 属性。我们在前面配置的信息可以被重写到XML文件之中如下：

```
<beans profile="dev"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="...">

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

```
<beans profile="production"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

当然，也可以通过嵌套 `<beans/>` 标签来完成定义部分：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <!-- other bean definitions -->

    <beans profile="dev">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
    </beans>
</beans>
```

`spring-bean.xsd` 已经被定义好，可以使用上面例子之中的这类标签。这将为XML文件的配置提供更多便利。

激活profile

现在，我们已经更新了配置信息来使用环境抽象，但是我们还需要告诉Spring来激活具体哪一个 `Profile` 。如果我们直接启动应用的话，现在就回抛出 `NoSuchBeanDefinitionException` 异常，因为容器会找不到Spring的Bean `dataSource` 。

有多种方法来激活一个Profile，最直接的方式就是通过编程的方式来直接调用 `Environment` API， `ApplicationContext` 中包含这个接口：

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("dev");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

额外的，Profile还可以通过 `spring.profiles.active` 中的属性来指定，可以通过系统环境变量，JVM系统变量，servlet上下文中的参数,甚至是JNDI的一个参数等来写入。在集成测试中，激活Profile可以通过 `spring-test` 中的 `@ActiveProfiles` 来实现。

需要注意的是，Profile的定义并不是一种互斥的关系，我们完全可以在同一时间激活多个Profile的。编程上来说，为 `setActiveProfile()` 方法提供多个Profile的名字即可：

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

也可以通过 `spring.profiles.active` 来指定 ` , ` 逗号分隔的多个Profile的名字：

```
-Dspring.profiles.active="profile1,profile2"
```

默认profile

默认的Profile就表示默认启用的Profile。参考如下代码：

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

如果没有其他的Profile被激活，那么上面代码定义的 `dataSource` 就会被创建，这种方式就是为默认情况下提供Bean定义的一种方式。一旦任何一个Profile激活了，默认的Profile则不会激活。

默认的Profile的名字可以通过 `Environment` 中的 `setDefaultProfiles()` 方法或者是通过 `spring.profiles.default` 属性来更改。

属性源抽象

Spring的 `Environment` 的抽象提供了一些层次化的搜索选项，来配置的源信息。具体的内容，参考如下代码：

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsFoo = env.containsProperty("foo");
System.out.println("Does my environment contain the 'foo' property? " + containsFoo);
```

在上面的代码片段之中，我们看到一个high-level的查找Spring是否定义 `foo` 属性的一种方式。为了知道Spring中是否包含这个属性， `Environment` 对象会针对 `PropertySource` 的集合进行查找。 `PropertySource` 是针对一些 `key-value` 的属性对的简单抽象，而Spring的 `StandardEnvironment` 是由两个 `PropertySource` 对象所组成的，一个代表的是JVM的系统属性（可以通过 `System.getProperties()` 来获取），而另一种则是系统的环境变量（通过 `System.getenv()` 来获取。）

> 这些默认的属性源都是 `StandardEnvironment` 的代表，在任何应用之中都可以使用。 `StandardServletEnvironment` 则是包含Servlet配置的环境信息，其中会包含很多Servlet的配置和Servlet上下文参数。 `StandardPortletEnvironment` 类似于 `StandardServletEnvironment` ，能够配置portlet上下文参数。可以参考其Javadoc了解更多信息。

具体的说，当使用 `StandardEnvironment` 的时候，调用 `env.containsProperty("foo")` 将返回一个 `foo` 的系统属性，或者是 `foo` 的运行时环境变量。

> 查询配置属性是按层次来查询的。默认情况下，系统属性优于系统环境变量，所以如果 `foo` 属性在两个环境中都有配置的话，那么在调用 `env.getProperty("foo")` 期间，系统属性值会优先返回。需要注意的是，属性的值是不会合并的，而是完全覆盖掉。 在一个普通的 `StandardServletEnvironment` 之中，查找的顺序如下，优先查找 `* ServletConfig` 参数（比如 `DispatcherServlet` 上下文），然后是 `* ServletContext` 参数（web.xml中的上下文参数），再然后是 `* JNDI` 环境变量，JVM系统变量（"-D"命令行参数）以及JVM环境变量（操作系统环境变量）。

最重要的是，整个的查找机制是可以配置的。也许开发者自己有些定义的配置源信息想集成到配置检索的系统中去。没问题，只要实现开发者自己的 `PropertySource` 并且将其加入到当前 `Environment` 的 `PropertySources` 之中即可：

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

在上面的代码之中， `MyPropertySource` 被添加到检索配置的第一优先级之中。如果存在一个 `foo` 属性，它将由于其他的 `PropertySource` 之中的 `foo` 属性优先返回。 `MutablePropertySources` API提供一些方法来允许精确控制配置源。

`@PropertySource` 注解

`@PropertySource` 注解提供了一种方便的机制来将 `PropertySource` 增加到Spring的 `Environment` 之中。 给定一个文件 `app.properties` 包含了key-value对 `testbean.name=myTestBean` ,下面的代码中，使用了 `@PropertySource` 调用 `testBean.getName()` 将返回 `myTestBean` :

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
public class AppConfig {
    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

任何的 `@PropertySource` 之中形如 `${...}` 的占位符，都可以被解析成 `Environment` 中的属性资源，比如:

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")
public class AppConfig {
    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

假设上面的`my.placeholder`是我们已经注册到`Environment`之中的资源，举例来说，JVM系统属性或者是环境变量的话，占位符会解析成对象的值。如果没有的话，`default/path`会来作为默认值。如果没有指定默认值，而且占位符也解析不出来的话，就会抛出`IllegalArgumentException`。

占位符解析

从历史上来说，占位符的值是只能针对JVM系统属性或者环境变量来解析的。但是现在不是了，因为环境抽象已经继承到了容器之中，现在很容易通过容器将占位符解析集成。这意味着开发者可以任意的配置占位符：

- 开发者可以自由调整系统变量还有环境变量的优先级
- 开发者可以额外增加自己的属性源信息

具体的说，下面的XML配置不会在意`customer`属性在哪里定义，只有这个值在`Environment`之中有效即可：

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

不幸的是，Java标准的 `java.net.URL` 类和标准的URL前缀处理器都不能很好的提供对低级资源的访问。比如：没有一个标准的 `URL` 实现可以用来访问类路径下或者是相对于 `ServletContext` 的资源。尽管可以为专门的 `URL` 前缀注册新的处理程序（类似于诸如 `http` 之类的前缀的现有处理程序），但是这通常相当复杂，并且 `URL` 接口仍然缺少某些期望的功能，例如检查所指的资源是否存在的方法。

9.1 Introduction 介绍

Aspect-Oriented Programming (面相切面编程 AOP) 用另外的一种编程架构的思考来补充 Object-Oriented Programming (面相对象编程OOP)。OOP 主要的模块单元是 class (类)，而 AOP 是 aspect（切面）。切面使得诸如事务管理等跨越多个类型和对象的关注点模块化。（这样的关注点在 AOP 的字眼里往往被称为 crosscutting （横切关注点））

AOP 是 Spring 里面的主要的组件。虽然 Spring IoC 容器没有依赖 AOP，意味着你不想用的时候也无需使用 AOP，但 AOP 提供一个非常有用的中间件解决方案来作为 Spring IoC 的补充。

Spring 2.0 AOP

Spring 2.0 引入了一种更加简单并且更强大的方式来自定义切面，用户可以选择使用 schema-based（基于模式）的方式或者使用 @AspectJ 注解样式。这两种风格都完全支持 Advice（通知）类型和 AspectJ 的切入点语言，虽然实际上仍然使用 Spring AOP 进行 weaving (织入）。

本章主要讨论 Spring 2.0 对基于模式和基于 @AspectJ 的 AOP 支持。 Spring 2.0 完全保留了对 Spring 1.2 的向下兼容性，下一章 10. Spring AOP APIs 将讨论 Spring 1.2 API 所提供的底层的 AOP 支持。

Spring中所使用的AOP：

- 提供声明式企业服务，特别是为了替代 EJB 声明式服务。最重要的服务是 declarative transaction management（声明性事务管理）， 这个服务建立在Spring的抽象事务管理（transaction abstraction）之上。
- 允许用户实现自定义的切面，用 AOP 来完善 OOP 的使用。

如果你只打算使用通用的声明式服务或者预先打包的声明式中间件服务，例如 *pooling*（缓冲池），你可以不直接使用 *AOP* ，可以忽略本章大部分内容

9.1.1 AOP concepts 概念

首先让我们从定义一些重要的 AOP 概念开始。这些术语不是 Spring 特有的。 不幸的是，Spring 术语并不是特别的直观；如果 Spring 使用自己的术语，将会变得更加令人困惑。

- Aspect（切面）： 一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是企业级 Java 应用中一个关于横切关注点的很好的例子。 在 Spring AOP 中，切面可以使用通用类（ schema-based基于模式的风格） 或者在普通类中以 @Aspect 注解（@AspectJ 注解样式）来实现。
- Join point (连接点）： 在程序执行过程中某个特定的点，比如某方法调用的时候或者处

理异常的时候。 在 Spring AOP 中，一个连接点 总是 代表一个方法的执行。

- Advice（通知）： 在切面的某个特定的 join point（连接点）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。 通知的类型将在后面部分进行讨论。许多 AOP 框架，包括 Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。
- Pointcut（切入点）： 匹配 join point（连接点）的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。 切入点表达式如何和连接点匹配是 AOP 的核心：Spring 默认使用 AspectJ 切入点语法。
- Introduction（引入）： 声明额外的方法或者某个类型的字段。 Spring 允许引入新的接口（以及一个对应的实现）到任何被 advise（通知）的对象。 例如，你可以使用一个引入来使 bean 实现 IsModified 接口，以便简化缓存机制。（在 AspectJ 社区 ，Introduction也被称为 inter-type declaration（内部类型声明））
- Target object（目标对象）： 被一个或者多个 aspect（切面）所advise（通知）的对象。也有人把它叫做 advised （被通知） 对象。 既然 Spring AOP 是通过运行时代理实现的，这个对象永远是一个 proxied（被代理） 对象。
- AOP proxy（AOP代理）： AOP 框架创建的对象，用来实现 aspect contract（切面契约）（包括通知方法执行等功能）。 在 Spring 中，AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。
- Weaving（织入）： 把 aspect（切面）连接到其它的应用程序类型或者对象上，并创建一个被 advised （被通知）的对象。 这些可以在编译时（例如使用 AspectJ 编译器），类加载时和运行时完成。 Spring 和其他纯 Java AOP 框架一样，在运行时完成织入。

通知的类型：

- Before advice（前置通知）： 在某 join point（连接点）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
- After returning advice（返回后通知）： 在某 join point（连接点）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
- After throwing advice（抛出异常后通知）： 在方法抛出异常退出时执行的通知。
- After (finally) advice（最后通知）： 当某join point（连接点）退出的时候执行的通知（不论是正常返回还是异常退出）。
- 环绕通知（Around Advice）： 包围一个join point（连接点）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

环绕通知是最常用的一种通知类型。跟 AspectJ 一样，Spring 提供所有类型的通知，我们推荐你使用尽量简单的通知类型来实现需要的功能。 例如，如果你只是需要用一个方法的返回值来更新缓存，虽然使用环绕通知也能完成同样的事情， 但是你最好使用 After returning 通知而不是环绕通知。 用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。 比如，你不需要调用 `JoinPoint`（用于Around Advice）的 `proceed()` 方法，就不会有调用的问题。

在Spring 2.0中，所有的通知参数都是静态类型，因此你可以使用合适的类型（例如一个方法执行后的返回值类型）作为通知的参数而不是使用一个对象数组。

pointcut（切入点）和 join point（连接点）匹配的概念是 AOP 的关键，这使得 AOP 不同于其它仅仅提供拦截功能的旧技术。 切入点使得 advice （通知）可独立于 OO 层次。 例如，一个提供声明式事务管理的around 通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

9.1.2 Spring AOP capabilities and goals 功能和目标

Spring AOP 用纯 Java 实现。它不需要专门的编译过程。Spring AOP 不需要控制类装载器层次，因此它适用于 Servlet 容器或应用服务器。

Spring 目前仅支持使用方法调用作为join point（连接点）（在 Spring bean 上通知方法的执行）。 虽然可以在不影响到 Spring AOP核心 API 的情况下加入对成员变量拦截器支持，但 Spring 并没有实现成员变量拦截器。 如果你需要通知对成员变量的访问和更新连接点，可以考虑其它语言，例如 AspectJ。

Spring 实现 AOP 的方法跟其他的框架不同。Spring 并不是要尝试提供最完整的 AOP 实现（尽管 Spring AOP 有这个能力）， 相反的，它其实侧重于提供一种 AOP 实现和 Spring IoC 容器的整合，用于帮助解决在企业级开发中的常见问题。

因此，Spring AOP 通常都和 Spring IoC 容器一起使用。 Aspect 使用普通的bean 定义语法（尽管 Spring 提供了强大的“autoproxying（自动代理）”功能）： 与其他 AOP 实现相比这是一个显著的区别。有些事使用 Spring AOP 是无法轻松或者高效的完成的，比如说通知一个细粒度的对象。 这种时候，使用 AspectJ 是最好的选择。不过经验告诉我们： 于大多数在企业级 Java 应用中遇到的问题，只要适合 AOP 来解决的，Spring AOP 都没有问题，Spring AOP 提供了一个非常好的解决方案。

Spring AOP 从来没有打算通过提供一种全面的 AOP 解决方案来取代AspectJ。 我们相信无论是 proxy-based（基于代理 ）的框架比如说Spring AOP 亦或是 full-blown 的框架比如说是 AspectJ 都是很有价值的，他们之间的关系应该是互补而不是竞争的关系。 Spring 可以无缝的整合 Spring AOP，IoC 和 AspectJ，使得所有的 AOP 应用完全融入基于 Spring 的应用体系。 这样的集成不会影响 Spring AOP API 或者AOP Alliance API；Spring AOP保留了向下兼容性。下一章 10. Spring AOP APIs会详细讨论 Spring AOP API。

一个 *Spring Framework* 的核心原则是 *non-invasiveness(非侵袭性)*；这意味着你不应该在您的业务/域模型被迫引入框架特定的类和接口。然而，在一些地方，*Spring Framework* 可以让你选择引入 *Spring Framework* 特定的依赖关系到你的代码，给你这样选择的理由是因为在某些情况下它可能是更容易读或编写一些特定功能。*Spring Framework* （几乎）总是给你的选择：你可以自由的做出明智的决定，选择最适合您的特定用例或场景。

这样的选择与本章有关的是 *AOP* 框架（和 *AOP* 类型）选择。你可以选择*AspectJ* 和/或 *Spring AOP* ，你也可以选择 *@AspectJ* 注解式的方法或*Spring* 的 *XML* 配置方式。事实上，本章以介绍 *@AspectJ* 风格为先不应该被视为 *Spring* 团队倾向于 *@AspectJ* 的方法胜过在 *Spring* 的*XML* 配置方式。

见*9.4. Choosing which AOP declaration style to use*里面有更完整的每种风格的使用原因探讨。

9.1.3 AOP Proxies 代理

Spring缺省使用标准 JDK dynamic proxies（动态代理）来作为AOP的代理。这样任何接口（或者接口的 set）都可以被代理。

Spring 也支持使用 CGLIB 代理. 对于需要代理类而不是代理接口的时候CGLIB 代理是很有必要的。 如果一个业务对象并没有实现一个接口，默认就会使用 CGLIB。 此外，面向接口编程也是一个最佳实践，业务对象通常都会实现一个或多个接口。此外，还可以强制的使用 CGLIB，在那些（希望是罕见的）在你需要通知一个未在接口中声明的方法的情况下，或者当你需要传递一个代理对象作为一种具体类型到方法的情况下。

一个重要的事实是，Spring AOP 是 proxy-based (基于代理)。见第9.6.1，“理解 AOP 代理”理解这个含义